# Fourier Transforms



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

**Step 1.** (Evaluate) Compute ordered pair of point-value representations

$\{(x_i, p(x_i)) \mid 0 \le i \le 2n-1\}$ and

$\{(x_i, q(x_i)) \mid 0 \le i \le 2n-1\}$

**Step 2.** (Pointwise Multiply) Compute

$y_i = p(x_i) q(x_i)$ for $0 \le i \le 2n-1$.

**Step 3.** (Interpolate) compute

$r_0, r_1, \cdots, r_{2n-1}$ s.t.

$y_i = r(x_i)$ for $0 \le i \le 2n-1$

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 $a+ib$: $|A|e^{i\alpha}$ where $|A| = \sqrt{a^2+b^2}$

$$\alpha = \cos^{-1}\frac{a}{\sqrt{a^2+b^2}}$$

$$= \tan^{-1}\frac{b}{a}$$

## Roots of unity:

$\omega^n = 1$, $\omega$ is the $n^{th}$ root of 1.

$\omega_n = e^{i\frac{2\pi}{n}}$ is the principle $n^{th}$ root of 1.

$\omega_n^0, \omega_n^1 = \omega_n, \omega_n^2, \omega_n^3, \ldots \omega_n^{n-1}$

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. $\omega_{dn}^{dk} = \omega_n^k$ because $\omega_{dn}^{dk} = e^{i\frac{2\pi dk}{dn}} = e^{i\frac{2\pi}{n}} = \omega_n^k$

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

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

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

**Proof(4):**

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

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

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

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 $j = 0, 1, \cdots n-1$

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

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

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

- The vector $y = (y_0, y_1, \cdots, y_{n-1})$ is the Discrete Fourier Transform of

$$p = (p_0, p_1, \cdots, p_{n-1})$$
$$y = DFT(p).$$

- The FFT is an algorithm to compute DFT quickly.
- We will see that same algorithm can be used for interpolation $DFT^{-1}$ 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 $n^{1.58}$ time algorithm, the FFT uses divide-and-conquer, but in a different way. Assume $n = 2^k$.

$$p^{[0]}(x) = p_0 + p_2 x + p_4 x^2 + \cdots + p_{n-2} x^{\frac{n-2}{2}}$$

$$p^{[1]}(x) = p_1 + p_3 x + p_5 x^2 + \cdots + p_{n-1} x^{\frac{n-2}{2}}$$

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

In order to evaluate $p(x)$ at the $n$ $n$th roots of unity, we just combine the evaluated values of $p^{[0]}(x)$ and $p^{[1]}(x)$ at these roots.

Suppose we have

$$y^{[0]} = DFT(p^{[0]}) \text{ and } y^{[1]} = DFT(p^{[1]}).$$

This means that

$$y^{[0]}_k = p^{[0]}(\omega^k_{n/2}), \text{ for } 0 \le k \le \frac{n-2}{2}, \text{ and}$$

$$y^{[1]}_k = p^{[1]}(\omega^k_{n/2}), \text{ for } 0 \le k \le \frac{n-2}{2}.$$

We can now compute $y = DFT(p)$ as:

For $0 \le k \le \frac{n-2}{2}$,

$$y_k = p(\omega^k_n) = p^{[0]}(\omega^{2k}_n) + \omega^k_n p^{[1]}(\omega^{2k}_n).$$

$$= y^{[0]}_k + \omega^k_n y^{[1]}_k$$

$$y_{\frac{n}{2}+k} = p(\omega^{\frac{n}{2}+k}_n)$$

$$= p^{[0]}(\omega^{n+2k}_n) + \omega^{n/2+k}_n p^{[1]}(\omega^{n+2k}_n)$$

$$= p^{[0]}(\omega^{2k}_n) - \omega^k_n p^{[1]}(\omega^{2k}_n)$$

$$= y^{[0]}_k - \omega^k_n y^{[1]}_k.$$

function FFT (p: polynomial) : point-values;

n := n(p); if n = 1 then return p

else

$p^{[0]} := (p_0, p_2, \cdots p_{n-2})$;

$p^{[1]} := (p_1, p_3, \cdots, p_{n-1})$;

$y^{[0]} := FFT(p^{[0]})$;

$y^{[1]} := FFT(p^{[1]})$;

.... line 1 .. $\omega := 1$; $\omega_n = e^{i\frac{2\pi}{n}}$;

for $k := 0$ to $\frac{n-2}{2}$ do

$y_k := y_k^{[0]} + \omega y_k^{[1]}$

$y_{\frac{n}{2}+k} := y_k^{[0]} - \omega y_k^{[1]}$;

.... line 2 ..... $\omega := \omega . \omega_n$

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 FFT($p$) and FFT($q$) we get a point-value representation of $r(x) = p(x)\,q(x)$ by multiplying the corresponding point-values:

$$y_j = p(\omega_n^j) \cdot q(\omega_n^j)$$

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 nth roots of unity. Recall that DFT can be written as follows:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ \vdots \\ r_{n-1} \end{bmatrix}$$

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

**Claim**

$$V_n^{-1} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \cdots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \cdots & \omega_n^{-(n-1)(n-1)} \end{bmatrix}$$

**Proof.** We need to show that $V_n^{-1}$ multiplied with $V_n$ gives the unit matrix.

$$V_n^{-1} \cdot V_n = \frac{1}{n}\left(\sum_{j=0}^{n-1} \omega_n^{-jk} \cdot \omega_n^{jl}\right)_{\substack{k=0,1,\ldots,n-1 \\ l=0,1,\ldots,n-1}}$$

$$= \frac{1}{n}\left(\sum_{j=0}^{n-1} \omega_n^{j(l-k)}\right)_{\substack{k=0,1,\ldots,n-1 \\ l=0,1,\ldots,n-1}}$$

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

So, $r = DFT^{-1}(y)$ is given by

$$r_k = \frac{1}{n} \sum_{j=0}^{n-1} \omega_n^{-jk}\, y_j$$

Compare it: $y_k = \sum_{j=0}^{n-1} \omega_n^{jk}\, p_j$

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

We have:

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

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

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

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

So, we compute recursively

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

This means

$$nr^{[0]} = y^{[0]}(\omega_{n/2}^{-k}) \text{ for } 0 \le k \le \frac{n-2}{2}, \text{ and}$$

$$nr^{[1]} = y^{[1]}(\omega_{n/2}^{-k}) \text{ for } 0 \le k \le \frac{n-2}{2}.$$

So, we can compute $r = DFT^{-1}(y)$:

For $0 \le k \le \frac{n-2}{2}$:

$$nr_k = y(\omega_n^{-k}) = y^{[0]}(\omega_n^{-2k}) + \omega_n^{-k} y^{[1]}(\omega_n^{-2k})$$
$$= nr_k^{[0]} + \omega_n^{-k} n r_k^{[1]}.$$

$$nr_{\frac{n}{2}+k} = y(\omega_n^{-(\frac{n}{2}+k)}) = y^{[0]}(\omega_n^{-2k}) + \omega_n^{-(\frac{n}{2}+k)} y^{[1]}(\omega_n^{-2k})$$
$$= nr_k^{[0]} - \omega_n^{-k} \cdot nr_k^{[1]}.$$

From the previous page,

$$n r_k = n r_k^{[0]} + \omega_n^{-k} n\, r_k^{[1]}$$

$$r_k = r_k^{[0]} + \omega_n^{-k} r_k^{[1]}.$$

$$n r_{\frac{n}{2}+k} = n r_k^{[0]} - \omega_n^{-k} \cdot n \cdot r_k^{[1]}$$

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

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

in line (1): $\omega_n = e^{i\frac{2\pi}{n}} \longrightarrow \omega_n^{-1} := e^{-i\frac{2\pi}{n}}$

in line (2): $\omega := \omega \cdot \omega_n \longrightarrow \omega := \omega \cdot \omega_n^{-1}$.

We conclude that both DFT and $DFT^{-1}$ 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,

$r = DFT_{2n}^{-1}(DFT_{2n}(p) \cdot DFT_{2n}(q))$ is such that $r(x) = p(x)q(x)$ and it can be computed in ~~$O(n)$~~ $O(n \log n)$ time.