Fast Fourier Transform

Posted on October 13, 2017

আজকে Fast Fourier Transform নিয়ে লিখব। Fast Fourier Transform মূলত Polynomial Multiplication এর জন্য ব্যবহার করা হয়। এই Tutorial এর মূল উদ্দেশ্য আসলে সেটাই।
শুরুতে কিছু Formal Defination দেখা যাক -

Discrete Fourier Transform

DFT কে এইভাবে Define করা যায় -

- DFT একটা Function, যেটা $\mathbb{C}^n$ থেকে $\mathbb{C}^n$ এ সংজ্ঞায়িত।
- এইটা $[a_0, a_1, a_2, \cdots, a_{n-1}]$ কে $[A_0, A_1, A_2, \cdots, A_n]$ এ রূপান্তর করে যেখানে -

$$A_j = \sum_{k=0}^{n-1} a_k \omega_n^{jk}$$

- এখানে ω_n হল যেকোনো **Primitive n^{th} root of unity**।
- সংক্ষেপে Primitive n^{th} root of unity হল এমন কোন সংখ্যা x যেন $x^n = 1$ হয়, কিন্তু অন্য যেকোনো $0 < k < n$ এর জন্য $x^k \neq 1$ হতে হবে। যেমন, চারটা 4^{th} root of unity আছে - $[1, -1, i, -i]$। কিন্তু এর মধ্যে Primitive হল শুধু i আর $-i$।

Fast Fourier Transform

Fast Fourier Transform হল একটা Algorithm যেইটা DFT কে Fast Calculate করতে পারে। 🙂
এই হল কিছু সংজ্ঞা। এইবার শুরু করা যাক এইগুলা দিয়ে কি করে, খায় না মাথায় দেয় ইত্যাদি।

DFT কেন গুরুত্বপূর্ণ?

আবার DFT এর Transformation এর Formula টা দেখা যাক -

$$A_j = \sum_{k=0}^{n-1} a_k \omega_n^{jk}$$

এইটা দেখেই অনেক বিদঘুটে লাগতে পারে। আমরা সব গুলা a_k নিচ্ছি , তাকে আবার কোন Root of unity এর Power দিয়ে গুন করছি, আবার সব গুলাকে যোগ করে একটা মাত্র সংখ্যা পাচ্ছি। এই কাজ আবার n বার করতে হচ্ছে।

কেন Primitive root of unity নিতে হবে? এতকিছু করে আমাদের লাভ কি? এর বিশেষত্ব কি?

DFT Special কারণ DFT কে সহজেই Inverse করা যায়। 😉
মানে আমাদের $[A_0, A_1, A_2, \cdots, A_{n-1}]$ দেওয়া থাকলে Unique ভাবে $[a_0, a_1, a_2, \cdots, a_{n-1}]$ বের করা সম্ভব।

DFT Inversion Formula টা হল -

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} A_k \omega_n^{-jk}$$

লক্ষ্য করি, DFT এর Inverse Formula কিন্তু DFT এর Formula এর মতই! শুধু n^{th} Primitive root of unity না নিয়ে তার Inverse নেওয়া হয়েছে, পরে Result কে n দিয়ে ভাগ করলেই হল।

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} A_k \left(\omega_n^{-1}\right)^{jk}$$

অর্থাৎ, আমরা কোন Sequence এর DFT জানলে মুল Sequence ও বের করতে পারি। এইটাই মূলত DFT এর বিশেষত্ব।
আর আমরা যদি Primitive root of unity না নেই তাহলে এইটাকে Inverse করা সম্ভব না। সেইটা প্রমাণ করতে গিয়ে দেখব।
শুধু Claim করলেই তো হবে না। প্রমাণ করতে হবে। তাহলে Prove করা যাক এইটা।
[কারো Proof পড়ার ইচ্ছা না থাকলে নিচের প্যারাটা Skip করতে পারেন 😠]

Proof of DFT Inversion Formula

Lemma: আমরা প্রমাণ করব যে, **কোন** n^{th} **root of unity** $\omega_n \neq 1$ **হলে** $\sum_{k=0}^{n-1} w_n^k = 0$।
সংজ্ঞা অনুযায়ী আমরা জানি $\omega_n^n = 1$। তাহলে -

$$\Rightarrow \omega_n^n - 1 = 0$$
$$\Rightarrow (\omega_n - 1)(\omega_n^{n-1} + \omega_n^{n-2} + \cdots + \omega_n + 1) = 0$$

যেহেতু $\omega_n \neq 1$ তাই $\sum_{k=0}^{n-1} w_n^k = 0$.

এবার আমরা DFT Inversion Formula টা প্রমাণ করতে পারি -

$$\sum_{k=0}^{n-1} A_k \omega_n^{-jk}$$

$$\Rightarrow \sum_{k=0}^{n-1} \left(\sum_{r=0}^{n-1} a_r \omega_n^{rk} \right) \omega_n^{-jk}$$

$$\Rightarrow \sum_{r=0}^{n-1} a_r \sum_{k=0}^{n-1} \omega^{k(r-j)}$$

এখন শুধু $\sum_{k=0}^{n-1} \omega^{k(r-j)}$ কে ভাল করে Inspect করা যাক। ২টা Case এ ভাগ করি -

Case 1: $r = j$**:**
এইটা সোজা। $r = j$ হলে সব গুলা পদ $\omega_n^0 = 1$ হয়ে যায়। তাই $\sum_{k=0}^{n-1} \omega^{k(r-j)} = n.$
তাহলে

$$\sum_{r=j} a_r \sum_{k=0}^{n-1} \omega^{k(r-j)} = n a_j$$

Case 2: $r \neq j$**:**
লক্ষ্য করি, $\left(\omega_n^{r-j}\right)^n = (\omega_n^n)^{r-j} = 1^{r-j} = 1.$
তারমানে w_n^{r-j} ও একটা n^{th} root of unity।

w_n এর সংজ্ঞা অনুযায়ী কোন $0 < k < n$ এর জন্য $w_n^k \neq 1$ হবে আর $w_n^n = 1$ হবে।
এখানে, $0 \leq r, j < n \Rightarrow 0 <\mid r - j \mid < n$।

সংজ্ঞানুযায়ী, $w_n^{r-j} \neq 1$।

তাহলে w_n^{r-j} একটা n^{th} root of unity $\neq 1.$
এখন উপরের Lemma ব্যবহার করে বলা যায় -

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

Case 1 আর 2 Combine করে বলা যায় -

$$\sum_{r=0}^{n-1} a_r \sum_{k=0}^{n-1} \omega^{k(r-j)} = na_j$$

$$\Rightarrow \sum_{k=0}^{n-1} A_k \omega_n^{-jk} = na_j$$

$$\Rightarrow a_j = \frac{1}{n} \sum_{k=0}^{n-1} A_k \omega_n^{-jk}$$

DFT Inversion Formula Proved. 🙂

এবার একটু অন্য দিকে যাওয়া যাক।

Polynomials

DFT এর আপাত দৃষ্টি তে কোন উপকারিতা না থাকলেও Polynomials এর আছে। তাই Polynomial এর কিছু দেখা যাক -

সাধারণত High School এ Polynomial কে Coefficient Form এ লেখা হয়। মানে -

$P(x) = \sum_{k=0}^{n-1} a_k x^k = a_0 + a_1 x^1 + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$.
একে শুধু $[a_0, a_1, a_2, \cdots, a_{n-1}]$ দিয়েও প্রকাশ করা যায়।

এই পদ্ধতির সুবিধা-অসুবিধা হল -

- ২টা Polynomial $P(x)$ আর $Q(x)$ যদি n মাত্রার হয় তাহলে $O(n)$ এ যোগ করা যায়। শুধু Coefficient গুলা যোগ করলেই হল।
- কিন্তু $P(x)$ আর $Q(x)$ কে গুন করতে গেলে $O(n^2)$ লেগে যাবে।

তাহলে এই Polynomial Represent করলে আমরা Efficiently Polynomial গুন করতে পারব না। কিন্তু Polynomial Multiplication এর অনেক প্রয়োজনীয়তা আছে।

তাই আরেকটা Polynomial Represent করার পদ্ধতি দেখা যাক।

Point-Value Form

Point Value Representation এ একটা Polynomial কে কতগুলা Point এ Evaluate করা হয়।
Formal ভাবে বলতে গেলে -

একটা Polynomial $P(x)$ এর Degree $n-1$ হলে এর Point-Value Representation হবে -

1. At Least n টা Point-Value Pair এর Set $[(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})]$।

2. যেখানে $y_i = P(x_i)$।
3. x_i গুলা Distinct।

Claim হলও n টা Point Value Pair Unique ভাবে যেকোনো $< n$ Degree Polynomial কে Represent করে। এইটা Prove করা সোজা।

Hint: আমাদের n টা Point-Value Pair দেওয়া আছে, আমাদের Target হল Polynomial টার Coefficient গুলা মানে $[a_0, a_1, \cdots, a_{n-1}]$ বের করা। n টা Coefficient আছে। আমদের n টা Point Value Pair দেওয়া থাকলে আমরা n টা Equation পাব। n টা Variable আর n টা Equation। তাই একটা মাত্র Solution থাকা সম্ভব খালি।

এইবার দেখা যাক Point Value Representation এ যোগ বিয়োগ গুণ করা যায় কিভাবে।

Point Value Form এ যোগ করা: মনে করি আমাদের ২টা $(n-1)$ Degree Polynomial $P(x)$ আর $Q(x)$ এর n টা Point Value Pair দেওয়া আছে। ২টা Polynomial কেই একই n টা Point এ Evaluate করা হয়েছে।
$P(x) \implies [(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})]$
$Q(x) \implies [(x_0, y'_0), (x_1, y'_1), \cdots, (x_{n-1}, y'_{n-1})]$।

তাহলে $R(x) = P(x) + Q(x)$ এর Point Value Representation বের করা যাক -

$R(x_0) = P(x_0) + Q(x_0) = y_0 + y'_0$।
অনুরূপভাবে বাকি x_i গুলার জন্যও বের করা যাবে।

তাহলে $R(x) \implies [(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \cdots, (x_{n-1}, y_{n-1} + y'_{n-1})]$
যেহেতু $R(x)$ ও একটা $(n-1)$ Degree Polynomial তাই এই n টা Point ই যথেষ্ট।
তাহলে Point Value Form এ Polynomial Add করতে $O(n)$ Time লাগল।

Point Value Form এ গুন করা: মনে করি আমাদের $R(x) = P(x)Q(x)$ বের করতে হবে। যদি $P(x)$ আর $Q(x)$ $(n-1)$ Degree Polynomial হয়, তাহলে $R(x)$ হবে $(2n-2)$ Degree Polynomial। তাহলে $R(x)$ এর জন্য আমাদের $(2n-1)$ টা Point লাগবে।
এর জন্য আমরা যা করতে পারি, $P(x)$ আর $Q(x)$ কে $(2n-1)$ টা Point এ Evaluate করতে পারি -
$P(x) \implies [(x_0, y_0), (x_1, y_1), \cdots, (x_{2n-2}, y_{2n-2})]$
$Q(x) \implies [(x_0, y'_0), (x_1, y'_1), \cdots, (x_{2n-2}, y'_{2n-2})]$।

সহজেই বুঝা যায় যে, $R(x_0) = P(x_0)Q(x_0) = y_0 y'_0$
তাহলে, $\Rightarrow R(x) \implies [(x_0, y_0 y'_0), (x_1, y_1 y'_1), \cdots, (x_n, y_{2n-2} y'_{2n-2})]$

সুতরং আমরা যদি ২টা Polynomial এর Point Value Pair পাই তাহলে আমরা $O(n)$ Time এ তাদের গুন করতে পারি!

তার মানে কি এই যে আমাদের ২টা Polynomial দেওয়া থাকলেই আমরা $O(n)$ Time এ Multiply করতে পারব? **না**! কারণ এর কিছু খারাপ দিক আছে। যেমন -

Transformation: আমরা দেখলাম যে ২টা Polynomial এর Point Value Form থেকে দ্রুত গুন করা যায়। ধরি আমাদের একটা Polynomial দেওয়া আছে।

$$P(x) = \sum_{k=0}^{n-1} a_k x^k$$

একে Point Value Form এ নিতে গেলে আমাদের n টা Point এ একে Evaluate করতে হবে। আর বারবারই সব গুলা Coefficient এর উপরে Loop চালাতে হবে। তাহলে Complexity হয়ে যাবে $O(n^2)$ 😟

Inverse Transformation: মনে করি আমরা কোনভাবে Point Value Form এ Transform করতেও পারলাম Fast. এখন সেই Point Value গুলা থেকে আসল Polynomial উদ্ধার করব কি করে?

এর জন্য আছে Lagrange Polynomial Interpolation.
আমাদের কিছু Point Value আছে ধরি $[(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})]$, তাহলে Polynomial টা হবে -

$$P(x) = \sum_{k=0}^{n-1} y_k \prod_{j=0, j\neq k}^{n-1} \frac{x - x_j}{x_k - x_j}$$

একটা উদাহরণ দেওয়া যাক।
একটা Polynomial এর Point Value Form - $[(0, 1), (1, 0), (2, 3), (3, -1)]$।
তাহলে Polynomial টা হল -

$$\begin{aligned} P(x) = \\ 1 \cdot \frac{(x-1)(x-2)(x-3)}{(0-1)(0-2)(0-3)} + \\ 0 \cdot \frac{(x-0)(x-2)(x-3)}{(1-0)(1-2)(1-3)} + \\ 3 \cdot \frac{(x-0)(x-1)(x-3)}{(2-0)(2-1)(2-3)} + \\ (-1) \cdot \frac{(x-0)(x-1)(x-2)}{(3-0)(3-1)(3-2)} \\ \Rightarrow P(x) = \frac{1}{6}(-11x^3 + 45x^2 - 40x + 6) \end{aligned}$$

কিন্তু এইক্ষেত্রেও আমাদের $O(n^2)$ লেগে যাবে। 😟
তাই আমাদের আর Fast কোন পদ্ধতি বের করতে হবে। এইখানেই আসে Fast Fourier Transform আর DFT। 😉

এইবার আসল জায়গায় যাওয়া যাক, কিভাবে আমরা দ্রুত Polynomial Multiplication করব Fast.

Polynomial Multiplication

আমরা একটা রাফ Algorithm দেখি ২টা Polynomial গুণ করার জন্য। ধরে নেই উভয়ে $(n-1)$ Degree Polynomial। কোনটার Degree কম হলে শেষে কতগুলা 0 Coefficient যোগ করে সমান করে নেওয়া যাবে।

ধরে নেই $P(x) = [a_0, a_1, \cdots, a_{n-1}]$ আর $Q(x) = [b_0, b_1, \cdots, b_{n-1}]$.

1. শুরুতে $P(x)$ আর $Q(x)$ কে $2(n-1)+1 = 2n-1$ টা Point এ Evaluate করতে হবে।
2. এবার Point Value Form এ তাদের গুণ করতে হবে।
3. এর পরে নতুন Point Value Pair গুলাকে Inverse Transform করলেই কাজ শেষ।
4. শুরুতে আমরা $2n-1$ টা Point এ Evaluate করেছিলাম, কারণ $R(x) = P(x)Q(x)$ একটা $2n-2$ degree polynomial. তাই Point-value form এ অন্তত $2n-1$ টা Pair থাকা লাগবেই।

এখানে 1 আর 3 কিন্তু আমরা এখনো দ্রুত করতে পারি না।
আচ্ছা, এবার আরেকবার DFT এর দিকে দেখা যাক।

DFT $[a_0, a_1, \cdots, a_{n-1}]$ কে $[A_0, A_1, \cdots A_{n-1}]$ এ পরিবর্তন করে যাতে -

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

আরেকবার আমাদের Polynomial টা দেখা যাক -

$$P(x) = \sum_{k=0}^{n-1} a_k x^k$$

DFT আর এই Polynomial এর মধ্যে কি মিল দেখা যায়? DFT আসলে কি করছে?
ভাল করে লক্ষ্য করলে দেখা যাচ্ছে যে DFT আসলে এই $P(x)$ কেই n টা Point এ Evaluate করে আউটপুট দিচ্ছে!
আর সেই n টা Point হল $[\omega_n^0, \omega_n^1, \omega_n^2, \cdots \omega_n^{n-1}]$
বা অন্যভাবে বললে $A_j = P(\omega_n^j)$

তাহলে এখন আমরা উপরের Algorithm এর প্রথম ধাপের Polynomial Evaluate করার বদলে আরেকটা কাজ করতে পারি -
আগে আমাদের Coefficient এর Array কে Resize করে n থেকে $(2n-1)$ করব, শেষে 0 যোগ করে।
এইবার আমরা $[a_0, a_1, \cdots, a_{2n-2}]$ Array টাকে DFT তে পাঠিয়ে দেব। তাহলে DFT আরেকটা Array $[A_0, A_1, \cdots, a_{2n-2}]$ Return করবে।
তাহলে আমাদের $P(x)$ এর Point Value Form হবে -

$$P(x) \Rightarrow [(\omega_{2n-2}^0, A_0), (\omega_{2n-2}^1, A_1), \cdots, (\omega_{2n-2}^{2n-2}, A_{2n-2})]$$

আর তৃতীয় ধাপে আমাদের কাছে $[A_0, A_1, \cdots, A_{2n-2}]$ থাকবে, এর Inverse DFT নিলেই আমরা আসল Polynomial এর Coefficient গুলা পেয়ে যাব।

তাহলে এখন খালি বাকি DFT কিভাবে Fast Calculate করা যায়। 😉

Calculating DFT Fast - Fast Fourier Transform

আরেকবার DFT এর Defination কে দেখা যাক একটু অন্য ভাবে।
আমাদের Polynomial $F(x) \Rightarrow [a_0, a_1, \cdots, a_{n-1}]$ দেওয়া থাকবে, আমাদের $[A_0, A_1, \cdots A_{n-1}]$ বের করতে হবে যাতে $A_j = F(\omega_n^j)$ হয়, বা কঠিন করে বললে -

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

সুবিধার জন্য আমরা ধরে নেব n একটা 2 এর Power।
এবার কিছু Observe করা যাক।

Observation 1: আমরা n টা Point এ Polynomial কে Evaluate করছি -

$$[1, \omega_n, \omega_n^2, \cdots, \omega_n^{n/2-1}, \omega_n^{n/2}, \cdots, \omega_n^{n-1}]$$

মাঝখানের জায়গাটা আলাদা করে দেখানোর কারণ হল ওই $\omega_n^{n/2}$ তে একটা জিনিস লুকিয়ে আছে।
আমরা জানি, $\omega_n^n = 1 \Rightarrow \left(\omega_n^{n/2}\right)^2 = 1$
আবার, ω_n হল এমন সংখ্যা যেন যেকোনো $0 < k < n$ এর জন্য $\omega_n^k \neq 1$ হয়।
তাহলে উপরের Equation এ $\omega_n^{n/2} \neq 1$
সুতরাং, $\left(\omega_n^{n/2}\right)^2 = 1 \Rightarrow \omega_n^{n/2} = -1$

তাহলে আমরা আসলে এই Point গুলাতে Evaluate করছি বলা যায় -

$$[1, \omega_n, \omega_n^2, \cdots, \omega_n^{n/2-1}] \mid [-1, -\omega_n, -\omega_n^2, \cdots, -\omega_n^{n/2-1}]$$

Observation 2: Observation 1 এর পরে আমাদের পরবর্তি কাজ কি হওয়া উচিত সেইটা Obvious।
আমাদের দেখতে হবে আমরা $P(x)$ জানলে কোনভাবে দ্রুত $P(-x)$ বের করতে পারি নাকি।
একটা চেষ্টা করা যাক। ধরি,

$$P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 = (a_0 + a_2x^2) + x(a_1 + a_3x^2)$$
$$P(-x) = a_0 - a_1x + a_2x^2 - a_3x^3 = (a_0 + a_2x^2) - x(a_1 + a_3x^2)$$

আশাকরি এইটা Self Explanatory। এইটাকে Generalize করা যায় এইভাবে -
আমরা ২টা নতুন Polynomial বানাব -

$$F(x) = [a_0, a_2, a_4, \cdots] = a_0 + a_2x + a_4x^2 + \cdots$$
$$G(x) = [a_1, a_3, a_5, \cdots] = a_1 + a_3x + a_5x^2 + \cdots$$

তাহলে আমরা $P(x)$ আর $P(-x)$ কে এর সাপেক্ষে লিখতে পারি -

$$P(x) = F(x^2) + xG(x^2)$$
$$P(-x) = F(x^2) - xG(x^2)$$

Observation 3: সবচেয়ে গুরুত্তপুর্ণ জায়গা। সম্পুর্ণ না বুঝা পর্যন্ত পড়ার অনুরধ রইল।

আমরা $P(x)$ কে $[1, \omega_n, \omega_n^2, \cdots, \omega_n^{n/2-1}, -1, -\omega_n, -\omega_n^2, \cdots - \omega_n^{n/2-1}]$ Point গুলাতে Evaluate করতে চাচ্ছি।
Observation 2 থেকে দেখছি $P(x) = F(x^2) + xG(x^2)$। Positive x গুলার জন্য। আর Negative গুলার জন্য $P(-x) = F(x^2) - xG(x^2)$।

এইখানে লক্ষ্য করার বিষয় হল আমাদের x Positive হোক বা Negative হক, আমাদের দরকার কিন্তু $F(x^2)$ আর $G(x^2)$ এর মান গুলা।

তার মানে আমাদের F আর G কে $[(1)^2, (\omega_n)^2, \left(\omega_n^2\right)^2, \cdots] = [1, \omega_n^2, \omega_n^4, \omega_n^6, \cdots]$ Point গুলাতে Evaluate করতে হবে। তাহলে আমরা আবার P Reconstruct করতে পারব Observation 2 এর মত করে।

কিন্তু এই সব $[1, \omega_n^2, \omega_n^4, \omega_n^6, \cdots]$ Point এ কিভাবে আমরা F আর G কে Evaluate করব?

এইখানে আরেকটা জিনিস প্রমাণ করার আছে।
Claim: $\omega_n^{2k} = \omega_{n/2}^k$
Proof:

$$\omega_n^{2k} = \left(\omega_n^2\right)^k$$

এখন $w = \omega_n^2$ এর মধ্যে একটা গুণ আছে। (w ধরে নিলাম)
সেটা হল - যেকোনো $0 < k < n/2$ হলে $w^k \neq 1$ [Proof is too simple]

সংজ্ঞা অনুসারে, $w = \omega_{n/2}$
বা, $\omega_n^2 = \omega_{n/2}$।

এইটাকে আগের সমীকরণে বসালেই মূল প্রমাণ সম্পন্ন হবে।

এখন, আমরা F আর G কে $[1, \omega_n^2, \omega_n^4, \omega_n^6, \cdots]$ Point গুলায় Evaluate করতে চাচ্ছিলাম।
কেবল যেই প্রমাণটা করলাম সেইটা থেকে বলা যায় এই Point গুলা আসলে এইটার Equilivant -

$$[\omega_{n/2}^0, \omega_{n/2}^1, \omega_{n/2}^2, \omega_{n/2}^3, \cdots]$$

দেখে কি পরিচিত মনে হয় না?
F আর G কিন্তু $(n/2-1)$ Degree Polynomial, আর এদের আমরা $[\omega_{n/2}^0, \omega_{n/2}^1, \omega_{n/2}^2, \omega_{n/2}^3, \cdots, \omega_{n/2}^{n/2-1}]$ Point গুলাতে Evaluate করতে চাচ্ছি!!! এইটাই কিন্তু Exactly আবার DFT! 😉

Putting Everything Altogether

আমরা সহজেই FFT এর জন্য Recurrsive Function লিখতে পারি। প্রক্রিয়াটা এরকম হবে -

1. FFT Function একটা Polynomial $[a_0, a_1, a_2, \cdots, a_{n-1}]$ Input নেবে, আর তাদের $[\omega_n^0, \omega_n^1, \omega_n^2, \cdots, \omega_n^{n-1}]$ এ Evaluate করে $[A_0, A_1, \cdots, A_{n-1}]$ Return করবে। উল্লেখ্য যে n কে কিন্তু 2 এর Power হতে হবে।
2. Base Case: $n = 1$ হলে Polynomial এর Coefficient Array টাই Return করা যাবে।
3. নাহলে আমরা Polynomial টাকে F আর G তে ভাগ করব। F এ Even Position এর Coefficient গুলা থাকবে, আর G তে বাকিগুলা। ২ ভাগের জন্য আবার FFT Call করব।
4. ২ ভাগের জন্য FFT Call করলে সেইটা F আর G কে $[1, \omega_{n/2}^1, \omega_{n/2}^2, \cdots, \omega_{n/2}^{n/2-1}]$ Point গুলাতে Evaluate করে Return করবে।
5. Observation 2 Use করে আমরা P এর সব গুলা মান বের করতে পারব।
6. Value গুলা Calculate করা হলে সেগুলা Return করতে হবে।

Complexity Analysis

ধরি, $T(n)$ হল একটা n ডিগ্রি Polynomial এর Coefficient গুলার DFT করার জন্য Time Complexity.
আমরা n কে ২ ভাগে ভাগ করি এবং উভয় ভাগের জন্য আবার DFT করি। এইখানে $2T(n/2)$ লাগে।
এর পরে Result Combine করার জন্য $O(n)$ Time লাগে। তাহলে -
$T(n) = 2T(n/2) + O(n)$
সমাধান করে পাব $T(n) = O(n \log n)$

Sample Implementation

Implementation এ প্রথমেই মনে আসতে পার ω_n কে কিভাবে লিখব Code এ। আর Complex Number কে কিভাবে প্রকাশ করব...
Complex Number এর জন্য C++ এর STL `std::complex<T>` ব্যবহার করা যেতে পারে, অথবা Just একটা Struct বানিয়ে $a + bi$ রাখার জন্য `a, b` Variable আর Arithmetic Operation গুলা Define করা যেতে পারে।

Root's of unity সম্পর্কে বেশি বলা হয় নি। কারো এ সম্পর্কে না জানা থাকলে বা আরও জানতে ইচ্ছা করলে এইখানে দেখতে পারেন - Primitive Roots of Unity - Brilliant (https://brilliant.org/wiki/primitive-roots-of-unity/)
তবে এইটা জেনে রাখা যায় যে, একটা n এর জন্য $\phi(n)$ টা Primitive n^{th} root of unity পাওয়া যায়। আর সব গুলাকে এইভাবে লেখা যায় -

$$e^{2\pi ik/n} : 1 \leq k \leq n, gcd(k, n) = 1$$

আমরা Code করার সময় যেকোনো একটা নিলেই হবে। সহজ করার জন্য আমরা সবসময় $e^{2\pi i/n}$ নিতে পারি।

নিচে একটা Sample Implementation দেওয়া হল।

```
typedef complex <long double> Complex;
typedef valarray <Complex> ValComplex;

const long double PI = 2 * acos(0.0);

void fft(ValComplex &p, bool inverse = 0) {
  int n = p.size();
  if(n <= 1) return;

  ValComplex f = p[slice(0, n/2, 2)],
             g = p[slice(1, n/2, 2)];

  // splice(a, b, c) will return number in indexes
  // a, a + c, a + 2c, .... a + (b-1)c

  fft(f, inverse); fft(g, inverse); // FFT for F and G

  Complex omega_n = exp(Complex(0, 2 * PI / n)), w = 1;
  if(inverse) omega_n = Complex(1, 0) / omega_n;

  for(int k = 0; k < n / 2; k++) {
    // Here w = omega_n^k
    Complex add = w * g[k];
    // this is p(x)
    p[k]       = f[k] + add;
    // Note that p(-x) should be in (x+n/2)th position
    p[k + n/2] = f[k] - add;
    w *= omega_n;
  }
}
void ifft(ValComplex &p) {
  fft(p, 1);
  p /= p.size(); // Divide each element by p.size()
}
```

লক্ষ্য করলে দেখা যায় আমরা এই Implementation এ আসলে DFS Style এ করছি, মানে আগে Depth বেশি তে গেছি, কিন্তু এই Approach অনেক Time consuming আর Memory লাগে বেশি। এইটা আমরা BFS Style এও করতে পারি। এই ক্ষেত্রে আমরা আগে থেকেই Special Algorithm (Bit Reverse Copy বলা হয়) দিয়ে Leaf Node এ কি কি থাকার কথা সেইগুলা বের করে নেই। এর পরে এক লেভেল বাই লেভেল করে Combine Operation গুলা করি। এতে Extra Memory ও লাগেনা। পরে কোন পর্বে Optimization গুলা নিয়ে কথা বলব। তবে আপাতত কেউ এই Implementation এ TLE খেলে এই Code (https://github.com/RezwanArefin01/CodeTemplate/blob/master/Fast%20Fourier%20Transform.cpp) টা ব্যবহার করতে পারেন।

Back to Polynomial Multiplication

আরেকবার ২টা Polynomial $P(x)$ আর $Q(x)$ কে গুণ করার জন্য তাহলে আমরা এইরকম করতে পারি -

1. প্রথমে R = PQ এর Degree বের করি। মানে Result Store করতে যে জায়গা লাগবে আরকি।
2. R এর Degree কে Nearest 2 এর Power এ Round **UP** করতে হবে।
3. এবার P আর Q এর Coefficient গুলার Array এর শেষে প্রয়োজনে 0 Insert করে R এর সমান করতে হবে।
4. P, Q এর জন্য FFT করতে হবে, এর পরে Point-Value গুলা গুণ করতে হবে।
5. এর পরে Inverse FFT মারলেই শেষ। 🙂

নিচে একটা Sample Implementation দেওয়া হল -

```
vector<int> multiply(vector<int> a, vector<int> b) {
  int n = a.size(), m = b.size();
  int t = n + m - 1, sz = 1; // t is degree of R
  while(sz < t) sz <<= 1;  // rounding to nearest 2^x

  ValComplex x(sz), y(sz), z(sz);
  // Resize first polynomial by inserting 0.
  for(int i = 0; i < n; i++) x[i] = Complex(a[i], 0);
  for(int i = n; i < sz; i++) x[i] = Complex(0, 0);

  // Resize second polynomial by inserting 0.
  for(int i = 0; i < m; i++) y[i] = Complex(b[i], 0);
  for(int i = m; i < sz; i++) y[i] = Complex(0, 0);

  fft(x); fft(y);   // Do fft on both polynomial
  // Multiply in Point-Value Form
  for(int i = 0; i < sz; i++) z[i] = x[i] * y[i];

  ifft(z);  // Inverse FFT
  vector<int> res(sz);
  // Precision problem may occur, round to nearest integer
  for(int i = 0; i < sz; i++) res[i] = z[i].real() + 0.5;
  // remove trailing 0's
  while(res.size() > 1 && res.back() == 0) res.pop_back();
  return res;
}
```

Practice Problems

Problem 1:

SPOJ VFMUL (http://www.spoj.com/problems/VFMUL/): Very Basic Application. ২টা অনেক বড় Number দেওয়া আছে। এদের গুণ করতে হবে।
এইটা কিন্তু সাধারণ BigIntLibrary দিয়ে হবে না। কারণ প্রায় সব Small Library তে $O(n^2)$ Implementation দেওয়া থাকে। কিন্তু এই প্রবলেম এ সেইটা কাজ করবে না। কারণ এইখানে $|A| \leq 30000$। তাই Fast Algorithm লাগবে।
এইটা সহজেই করা যায় Number গুলাকে 10 এর Polynomial হিসাবে প্রকাশ করে। যেমনঃ

$$123 \times 43 = (3 + 2 \times 10 + 1 \times 10^2)(3 + 4 \times 10) = 9 + 18 \times 10 + 11 \times 10^2 + 4 \times 10^3$$

লক্ষ্য করার বিষয় হল, নতুন Polynomial এর কোন Coefficient সর্বচ্চ 81×30000 হতে পারে। তাই এই Coefficient গুলা আমরা FFT দিয়ে বের করে এর পরে Print করার সময় Carry রেখে একটা একটা করে Digit বের করতে পারব।

Problem 2:

SPOJ POLYMUL (http://www.spoj.com/problems/POLYMUL/): Just Implementation.

Problem 3:

একটা Array দেওয়া আছে, Size Atmost 10^5। Element গুলা Atmost $2^{16}-1$. কত গুলা Query আছে। একটা Number S দেওয়া হবে। বলতে হবে কয়টা Pair (i,j) আছে যেন $A_i + A_j = S$ হয়, $0 \leq i,j < n$।

এই প্রবলেম টার Solution টা সহজ। একটা Polynomial বানাই এইভাবে -

$$P(x) = x^{A_0} + x^{A_1} + x^{A_2} + \cdots + x^{A_{n-1}}$$

তাহলে, একটা Query S এর জন্য Answer হল $P(x) \times P(x)$ এর মধ্যে x^S এর সহগ যা তাই। 🙂 কেন এইটা Solution হবে সেইটা একটা উদাহরণ দিয়ে দেখানো যাক।

ধরি, Array টা হল $[1,2,3]$, তাহলে Polynomial হল $P(x) = x^1 + x^2 + x^3$।
এবার, $P(x) \times P(x)$ কে ভেঙ্গে দেখা যাক -

$$\begin{aligned}
&P(x) \times P(x) \\
&= (x^1 + x^2 + x^3)(x^1 + x^2 + x^3) \\
&= x^1\left(x^1 + x^2 + x^3\right) + x^2\left(x^1 + x^2 + x^3\right) + x^3\left(x^1 + x^2 + x^3\right) \\
&= \left(x^{1+1} + x^{1+2} + x^{1+3}\right) + \left(x^{2+1} + x^{2+2} + x^{2+3}\right) + \left(x^{3+1} + x^{3+2} + x^{3+3}\right)
\end{aligned}$$

দেখা যাচ্ছে যে এই Product এ All Pair Sum গুলাই Exponent আকারে আছে। তাই একটা নির্দিষ্ট Sum S কয়বার আছে সেইটা হল Just x^S এর Coefficient যা তাই। 🙂

$$\begin{aligned}
P(x) \times P(x) &= \left(x^2 + x^3 + x^4\right) + \left(x^3 + x^4 + x^5\right) + \left(x^4 + x^5 + x^6\right) \\
\Rightarrow P(x) &= x^2 + 2x^3 + 3x^4 + 2x^5 + x^6
\end{aligned}$$

এখানে x^4 এর Coefficient 3। তাই ৩টা Pair আছে যাদের Sum 4 - $(1,3), (2,2), (3,1)$।

Problem 4:

Toph - Counting Triplets (https://toph.co/p/counting-triplets): Medium। একটা Array দেওয়া আছে। কয়টা Triplet $< i,j,k >$ আছে জেন $A_i + A_j = A_k \text{ and } i < j < k$ হয়।
এইটা করতে Block Decomposition এর একটা Varient + FFT লাগে, Block Size ও Non-Standard। অবশ্য এইখানে Time Limit অনেক বেশি, তাই Highly Optimized $O(n^2)$ Pass করে। কারো এইটা কঠিন লাগলে আপাতত বাদ দিতে পারেন। এইটা Basic Problem না FFT এর।
Block Decomposition Series এর পরের কোন পর্বে এই Varient টা (Dynamic Graph Connectivity Problem বলে) নিয়ে লেখার চেষ্টা করব। 🙂

Problem 5:

CodeChef - COUNTARI (https://www.codechef.com/problems/COUNTARI): Toph এর Problem টার মতই। খালি $A_j - A_i = A_k - A_j$ হতে হবে।

ধন্যবাদ সবাইকে। 🙂

Category: Discrete Math (/category#Discrete%20Math)

(htt... (htt... (htt...

← PREVIOUS POST (/POSTS/BLOCK-DECOMPOSITION-01/)

NEXT POST → (/POSTS/CONVEX-HULL-TRICK/)