# Implementasi Fast Fourier Transform Dalam Kalkulasi Barisan Modular Dari Koefisien Ekspansi Binomial

Adril Putra Merin - 13522068[1]
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
*[1]13522068@std.stei.itb.ac.id*

***Abstract*—Fast Fourier transform adalah salah satu algoritma untuk melakukan perkalian dua buah polinomial dengan efisien. Algoritma ini memungkinkan pemecahan masalah kompleks yang dapat disajikan dalam bentuk polinomial menjadi lebih mudah dan cepat. Salah satu masalah yang dapat diselesaikan menggunakan algoritma ini adalah proses kalkulasi barisan modular dari koefisien ekspansi binomial. Persoalan ini dapat dimodelkan menjadi persoalan polinomial sehingga dapat diselesaikan menggunakan algoritma FFT. Makalah ini akan membahas aplikasi algoritma FFT dalam pemecahan masalah tersebut. Selain itu, makalah ini akan mencoba melakukan perbandingan antara algoritma FFT dengan algoritma naif atau sederhana.**

***Keywords*—teori bilangan, ekspansi binomial, fft, fast fourier transform, koefisien binomial, barisan.**

## I. PENDAHULUAN

Ekpansi binomial adalah salah satu teorema penting di bidang kombinatorial. Ekspansi binomial menjelaskan mengenai ekpansi dari sebuah ekspresi binomial dalam bentuk $(x+y)^n$. Menurut teorema ini, kita dapat mengekspansi bentuk $(x+y)^n$ menjadi:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k \quad (1)$$

Dalam persamaan tersebut, *n* adalah bilangan bulat non-negatif dan setiap $\binom{n}{k}$ adalah koefisien binomial yang merupakan bilangan bulat positif.

Definisikan $\{a_t\}$ sebagai barisan yang menyatakan koefisien dari ekspansi binomial dengan $0 \le t \le n$ sehingga dapat dituliskan $a_t = \binom{n}{t}$. Untuk *n* yang relatif kecil, perhitungan barisan tersebut masih dapat dilakukan di komputer pada umumnya. Namun, untuk *n* yang bernilai besar, misalkan $10^7$, hal ini menjadi hampir tidak mungkin dilakukan pada komputer biasa karena keterbatasan memori dan waktu pemrosesan.

Pada makalah ini, kita akan membatasi barisan $\{a_t\}$ diatas menggunakan *k*. Untuk itu, mari kita definisikan barisan $\{b_t\}$ dimana suku ke-*t* pada barisan tersebut adalah jumlah koefisien dari setiap suku pada $(1+x)^n$ yang derajatnya memenuhi persamaan $i \bmod 2^k = t$ untuk $0 \le t \le 2^k - 1$. Untuk mempermudah perhitungan, kita akan melakukan modulo terhadap bilangan prima 998244353 pada setiap hasil perhitungan pada barisan tersebut. Selain itu, kita akan memberikan batasan $k \le 17$ dan $n \le 998244353$. Berdasarkan deskripsi tersebut, dapat diperoleh definisi formal $b_t$ sebagai berikut:

$$b_t = \left( \sum_{i=0, i \bmod 2^k}^{n} \binom{n}{i} \right) \bmod 998244353 \quad (2)$$

Kita akan melihat bahwa solusi terbaik untuk masalah ini memerlukan algoritma yang cepat dan efisien untuk mengalikan polinom berderajat banyak. Algoritma *Fast Fourier Transform* (FFT) adalah salah satu algoritma yang dapat melakukan hal tersebut dengan efisien. Makalah ini akan membahas kompleksitas Algoritma FFT dan penerapannya pada masalah ini.

## II. DASAR TEORI

### A. Kombinatorial

Secara singkat, kombinatorial adalah salah satu cabang matematika yang berkaitan dengan cara penghitungan jumlah penyusunan objek-objek tanpa harus mengenumerasi semua kemungkinan susunannya[1].

Secara umum, terdapat dua kaidah paling dasar dalam melakukan enumerasi objek, yaitu kaidah penjumlahan dan kaidah perkalian. Misalkan terdapat *n* buah percobaan dimana banyak cara untuk melakukan percobaan ke-*i* adalah $p_i$ untuk setiap $i \in [1, n]$. Jika setiap percobaan saling lepas, maka menurut aturan penjumlahan, banyak cara untuk melakukan percobaan adalah $\sum_{i=1}^{n} p_i$. Perhatikan bahwa, setiap percobaan harus dilakukan secara eksklusif. Sebagai contoh, misalkan terdapat 3 buah kue, 2 buah eskrim, dan 4 buah permen. Menurut aturan penjumlahan, banyak cara untuk memilih kue, eskrim, atau coklat adalah:

$$P(K \text{ or } E \text{ or } C) = P(K) + P(E) + P(C) = 9$$

Disisi lain, misalkan terdapat $n$ buah percobaan dimana banyak cara untuk melakukan percobaan independen ke-$i$ adalah $p_i$ untuk setiap $i \in [1, n]$ dan kita ingin menghitung banyak cara untuk melakukan $n$ buah percobaan tersebut secara berurutan. Menurut aturan perkalian, banyak cara untuk melakukan $n$ buah percobaan tersebut secara berurutan adalah $\prod_{i=1}^{n} p_i$. Sebagai contoh, misalkan terdapat 3 buah cara untuk pergi dari kota P ke kota Q, 5 buah cara untuk pergi dari kota Q ke kota R, dan 2 buah cara untuk pergi dari kota R ke kota S. Maka banyak cara untuk pergi dari kota P ke S adalah $3 \times 5 \times 2 = 30$ cara.

### *B. Permutasi*

Permutasi adalah banyak cara untuk menyusun objek-objek dengan urutan yang berbeda. Permutasi adalah bentuk khusus dari kaidah perkalian[1]. Misalkan terdapat $n$ buah bola berbeda dan $k$ buah kotak dengan $k \leq n$, maka banyaknya cara untuk mengisi kotak-kotak tersebut dengan tepat 1 buah bola adalah

- Kotak ke-1 dapat diisi dengan salah satu dari $n$ bola.
- Kotak ke-2 dapat diisi dengan salah satu dari $n-1$ bola.
- Kotak ke-3 dapat diisi dengan salah satu dari $n-2$ bola.
  ...
- Kotak ke-$k$ dapat diisi dengan salah satu dari $n-k-1$ bola.

Menurut aturan perkalian, banyak cara untuk mengisi $k$ buah kotak dengan $n$ buah bola dimana setiap kotak tepat memiliki 1 bola adalah $n(n-1)(n-2)\ldots(n-k-1)$. Ekspresi ini dinotasikan sebagai $P(n,k)$ yang ekuivalen dengan

$$P(n,k) = n(n-1)(n-2)\ldots(n-k-1) = \frac{n!}{(n-k)!} \quad (3)$$

Perhatikan bahwa, untuk $k = n$, ekspresi tersebut menjadi $P(n,n) = n! = n \times (n-1) \times (n-2) \ldots \times 1$ yang dapat diinterpretasikan sebagai banyak cara untuk mengacak susunan $n$ buah objek.

### *C. Kombinasi*

Kombinasi adalah banyak cara untuk memilih kumpulan objek dari suatu himpunan objek dimana urutan kemunculan diabaikan. Kombinasi pada dasarnya adalah bentuk khusus dari kombinasi yang tidak mementingkan urutan kemunculan suatu objek[1].

Selanjutnya, kita akan mencoba menghubungkan permutasi dengan kombinasi dan menurunkan formula untuk kombinasi. Misalkan terdapat sebuah himpunan yang terdiri atas $n$ buah objek. Banyak cara untuk memilih $k$ objek dari himpunan tersebut adalah $P(n,k)$. Namun, karena urutan pemilihan objek tersebut diabaikan, maka kombinasi $k$ dari $n$ elemen adalah $P(n,k)/P(k,k)$. Jadi, kombinasi $k$ dari $n$ elemen dapat didefinisikan secara matematis sebagai berikut:

$$C(n,k) = \frac{P(n,k)}{P(k,k)} = \frac{n!}{(n-k)!\,k!} \quad (4)$$

Selain notasi di atas, kombinasi $k$ dari $n$ elemen juga dapat dinotasikan sebagai $\binom{n}{k}$ dan $C_k^n$.

### *D. Teorema Binomial*

Teorema binomial atau ekpansi binomial adalah teorema yang menjelaskan tentang ekpansi ekspresi binomial dalam bentuk $(x+y)^n$[2]. Menurut teorema binomial, untuk setiap bilangan bulat nonnegatif $n$, berlaku

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

Dalam persamaan tersebut, ekspresi $\binom{n}{k}$ adalah koefisien suku ke-$k$ yang merupakan bilangan bulat positif.

Dengan teorema binomial, kita dapat melakukan perhitungan kombinatorial kompleks dengan lebih efisien. Misalkan, kita ingin mencari berapa banyak cara untuk mengambil $k$ buah bola dari sebuah kumpulan bola sebanyak $n$. Secara matematis, masalah tersebut dapat dinyatakan sebagai:

$$P = \sum_{k=0}^{n} \binom{n}{k}$$

Perhatikan bahwa, dengan mensubstitusikan $x = 1$ dan $y = 1$ ke dalam persamaan (1), diperoleh persamaan sebagai berikut

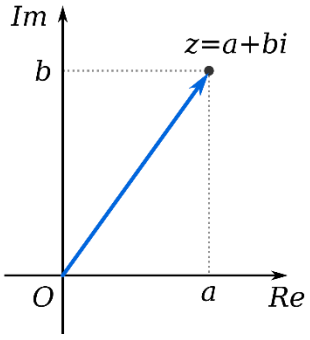
$$(1+1)^n = \sum_{k=0}^{n} \binom{n}{k} 1^{n-k} 1^k = \sum_{k=0}^{n} \binom{n}{k}$$

$$\therefore P = 2^n$$

### *E. Bilangan Kompleks*

Bilangan kompleks adalah sebuah sistem bilangan yang memperluas bilangan riil dengan elemen spesifik yang dinotasikan sebagai $i$. Bilangan $i$ adalah bilangan imajiner yang memenuhi persamaan $i^2 = -1$. Bilangan kompleks adalah bilangan yang dapat dinyatakan dalam bentuk $a + bi$ dimana $a$ dan $b$ adalah bilangan riil[3].

Secara visual, bilangan kompleks $z = a + bi$ dapat direpresentasikan sebagai sebuah tuple $(a, b)$ yang membentuk vector pada diagram Argand yang mewakili bidang kompleks.




**Gambar 1**. Diagram Argand

Sebuah bilangan kompleks $z = a + bi$ juga dapat dinyatakan dalam bentuk $z = r(\cos\theta + i\sin\theta)$. Bentuk ini disebut **bentuk polar**, dimana

$$r = \sqrt{a^2 + b^2}$$
$$\cos\theta = \frac{a}{r}$$
$$\sin\theta = \frac{b}{r}$$

Salah satu teorema yang berkaitan dengan bentuk polar suatu bilangan kompleks adalah Formula Euler yang secara matematis dinyatakan sebagai berikut,

$$e^{i\theta} = (\cos\theta + i\sin\theta) \quad (5)$$

Dengan mensubstitusikan persamaan *(5)* ke dalam bentuk polar bilangan kompleks sebelumnya, diperoleh

$$z = re^{i\theta}$$

### *F. Root of Unity (Akar Satuan)*

Root of unity atau, diterjemahkan secara bebas, akar satuan adalah bilangan kompleks sebarang yang menghasilkan 1 ketika dipangkatkan oleh suatu bilangan bulat *n*[4]. Konsep ini digunakan di banyak cabang matematika, salah satunya adalah discrete fourier transform yang akan kita bahas nanti.

Secara matematis, akar satuan ke-$n$ dengan $n > 0$ adalah sebuah bilangan $z$ yang memenuhi persamaan

$$z^n = 1. \quad (6)$$

Secara umum, bilangan $z$ tersebut dapat dianggap sebagai bilangan kompleks, jika tidak ada syarat tertentu. Dengan demikian, kita bisa mengaplikasikan formula Euler terhadap persamaan *(6)*.

$$z^n = 1 \Leftrightarrow z^n = e^{2k\pi i} \Leftrightarrow z = e^{\frac{2k\pi i}{n}}$$

Tinjau bahwa, fungsi trigonometri adalah sebuah fungsi periodik, sehingga nilai $e^{2k\pi i/n}$ akan terus berulang untuk setiap kelas residu modulo *n*.[3] Jadi, solusi yang diperoleh adalah *n* bilangan kompleks berbeda yang berbentuk $\omega_{n,k} = e^{2k\pi i/n}$ dengan $k = 0 \ldots n-1$. Selain itu, bilangan kompleks ini juga memiliki sifat yang menarik, salah satunya adalah bahwa akar utama satuan ke-*n* $\omega_n = \omega_{n,1} = e^{2\pi i/n}$ dapat digunakan untuk mendapatkan akar satuan ke-*n* lainnya: $\omega_{n,k} = (\omega_n)^k$[5].

### *G. Primitive Root (Akar Primitif) dan Fungsi Phi Euler*

Dalam aritmatika modular, sebuah bilangan $g$ disebut akar primitif modulo $n$ jika setiap bilangan yang koprima dengan $n$ kongruen terhadap pangkat $g$ modulo $n$. Secara matematis, $g$ adalah akar primitif modulo $n$ jika dan hanya jika untuk sebarang bilangan bulat $a$ dimana $\gcd(a, n) = 1$, terdapat bilangan bulat $k$ sedemikian rupa sehingga $g^k \equiv a \ (mod\ n)$[6].

Misalkan $g$ adalah sebuah akar primitif modulo $n$. Dapat ditunjukkan bahwa bilangan $k$ terkecil sedemikian rupa sehingga $g^k \equiv 1 \ (mod\ n)$ adalah $\varphi(n)$ *(Euler Totient Function)*. *Euler totient function* atau fungsi phi Euler yang diberikan oleh $\varphi(n)$ adalah fungsi yang menghitung banyak bilangan bulat positif hingga $n$ yang relatif prima terhadap $n$[7]. Secara matematis, fungsi phi Euler didefinisikan sebagai:

$$\varphi(n) = n\prod_{p|n}\left(1 - \frac{1}{p}\right) \quad (7)$$

Fungsi phi Euler tersebut adalah dasar dari teorema Euler. Menurut teorema Euler, jika $n$ dan $a$ adalah bilangan bulat positif yang saling koprima dan $\varphi(n)$ adalah fungsi phi Euler, maka $a$ pangkat $\varphi(n)$ kongruen dengan 1 modulo $n$[7]. Secara matematis, teorema Euler dapat dituliskan sebagai berikut:

$$a^{\varphi(n)} \equiv 1 \ (mod\ n) \quad (8)$$

Salah satu cara untuk mencari akar primitif modulo $n$ adalah dengan mengecek semua angka pada rentang $[1, n-1]$ satu-persatu apakah angka tersebut adalah akar primitif. Kompleksitas dari algoritma ini adalah $O(g.n)$ yang akan memakan terlalu terlalu banyak waktu sehingga kita perlu mencari alternatif algoritma lain.

Sebelumnya, sudah dibahas bahwa jika kita mengetahui bilangan bulat $k$ terkecil sehingga $g^k \equiv 1 \ (mod\ n)$ adalah $\varphi(n)$, maka $g$ adalah sebuah akar primitif modulo $n$. Berdasarkan teorema Euler, kita tahu bahwa $a^{\varphi(n)} \equiv 1 \ (mod\ n)$. Jadi, untuk mengecek apakah $g$ adalah sebuah akar primitif, kita hanya perlu memastikan bahwa untuk semua bilangan bulat $d \leq \varphi(n)$, berlaku $g^d \equiv 1 \ (mod\ n)$[6]. Terdapat beberapa optimasi yang dapat dilakukan terhadap algoritma ini, tetapi tidak akan dibahas pada makalah ini.

### *H. Discrete Fourier Transform (DFT)*

Misalkan terdapat sebuah polynomial berderajat $n-1$: $P(z) = a_0 z^0 + a_1 z^1 + \cdots + a_{n-1} z^{n-1}$. Pada implementasi sederhananya, polinomial tersebut dapat disimpan dalam struktur data *array* atau *vector* (di C++). Tanpa mengurangi keumuman, asumsikan bahwa $n$ adalah bilangan perpangkatan dua yang berbentuk $2^k$. Hal ini tentu tidak akan bermasalah karena kita hanya perlu menambahkan 0 pada *array* koefisien tersebut. Selanjutnya, definisikan *Discrete Fourier Transform* (DFT) dari $P(z)$ sebagai sebuah polinomial baru yang koefisiennya adalah $P(\omega_{n,k})$ untuk setiap $0 \leq k < n$ dengan $\omega_{n,k}$ adalah akar satuan ke- $n$ yang sudah dibahas sebelumnya. Secara matematis, DFT dari *vector* koefisien polynomial $P(z)$ dapat dinyatakan sebagai berikut

$$DFT(a_0, a_1, \ldots, a_{n-1}) = \left(P(\omega_{n,0}), P(\omega_{n,1}), \ldots, P(\omega_{n,n-1})\right)$$
$$= (P(\omega_n^0), P(\omega_n^1), \ldots, P(\omega_n^{n-1})) \quad (9)$$

Dengan cara yang sama, *Inverse Discrete Fourier Transform* (IDFT) didefinisikan sebagai invers DFT $P(z)$. Secara matematis, dapat dituliskan

$$IDFT\left(P\left(\omega_{n,0}\right), P\left(\omega_{n,1}\right), \ldots, P(\omega_{n,n-1})\right) = (a_0, a_1, \ldots, a_{n-1})$$

*( 10 )*

Salah satu penggunaan DFT yang sudah sempat disinggung sebelumnya adalah mengenai perkalian dua polinomial. Misalkan $A$ dan $B$ adalah polinomial. Kalkulasikan DFT terhadap masing-masing polinomial tersebut. Dengan mengalikan *vector* $DFT(A)$ dan $DFT(B)$, kita akan mendapatkan hasil berupa $DFT(A \cdot B)$ atau dengan kata lain:

$$DFT(A \cdot B) = DFT(A) \cdot DFT(B)$$

Selanjutnya, dengan mengaplikasikan inverse DFT, dapat diperoleh:

$$A \cdot B = IDFT(DFT(A) \cdot DFT(B))$$

## *I. Fast Fourier Transform (FFT)*

Fast Fourier Transform adalah sebuah algoritma untuk menghitung *Discrete Fourier Transform* (DFT) dari suatu polinomial derajat $N-1$ dengan kompleksitas waktu $O(Nlog(N))$. Perkembangan algoritma FFT dapat ditelusuri kembali hingga karya Carl Friedrich Gauss yang tidak dipublikasikan. Metode Gauss sangat mirip dengan apa yang dipublikasikan oleh James Cooley dan John Tukey pada tahun 1965, yang secara umum dianggap sebagai penemu algoritma FFT yang lebih modern dan umum[5].

Ide dasar dari FFT adalah tentang bagaimana mengaplikasikan metode *divide and conquer* dalam perhitungan DFT. Misalkan terdapat sebuah polinomial $P(z)$ derajat $n-1$ dengan $n = 2^k$ dan $n > 1$:

$$P(z) = a_0 z^0 + a_1 z^1 + \cdots + a_{n-1} z^{n-1}$$

Bagi polinomial $P(z)$ ke dalam dua polinomial yang lebih kecil dengan salah satu polinomial menyimpan koefisien dari suku genap dan polinomial lainnya menyimpan koefisien dari suku ganjil. Secara matematis, dapat dituliskan:

$$P_1(z) = a_0 z^0 + a_2 z^1 + \cdots + a_{n-2} z^{\frac{n}{2}-1}$$
$$P_2(z) = a_1 z^0 + a_3 z^1 + \cdots + a_{n-2} z^{\frac{n}{2}-1}$$

*( 11 )*

Tinjau bahwa,

$$P(z) = P_1(z^2) + zP_2(z^2)$$

*( 12 )*

Selanjutnya, kita akan mencari persamaan untuk $DFT(P)$. Tinjau suku ke-$k$ dari $DFT(P)$. Untuk $0 \le k < \frac{n}{2}$, kita dapat menggunakan hubungan pada *(11)* sehingga diperoleh:

$$P\left(\omega_{n,k}\right) = P_1\left(\omega_{n,k}\right) + \omega_{n,k} P_2(\omega_{n,k}) = P_1(\omega_n^k) + \omega_n^k P_2(\omega_n^k)$$

*( 13 )*

Tinjau nilai $k$ pada rentang $\frac{n}{2} \le k <$ n. Kita dapat memisalkan $k = \frac{n}{2} + m$ dengan $0 \le m < \frac{n}{2}$ sehingga diperoleh:

$$\begin{aligned} P\left(\omega_{n,k}\right) &= P_1(\omega_n^{2k}) + \omega_n^k P_2(\omega_n^{2k}) \\ &= P_1(\omega_n^{2m+n}) + \omega_n^{m+\frac{n}{2}} P_2(\omega_n^{2m+n}) \\ &= P_1(\omega_n^{2m}\omega_n^n) + \omega_n^m \omega_n^{\frac{n}{2}} P_2(\omega_n^{2m}\omega_n^n) \end{aligned}$$

Menurut teorema Euler pada *(5)*, $\omega_n^n = 1$ dan $\omega_n^{n/2}$=-1. Jadi, persamaan akhir yang diperoleh untuk DFT pada rentang $\frac{n}{2} \le k <$ n adalah:

$$P\left(\omega_{n,k}\right) = P_1(\omega_n^k) - \omega_n^k P_2(\omega_n^k)$$

*( 14 )*

## *J. Inverse Fast Fourier Transform*

Berdasarkan definisinya, kita dapat menyatakan proses perhitungan FFT dalam bentuk matriks sebagai berikut:

$$\begin{pmatrix} w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & \cdots & w_n^{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

*( 15 )*

Matriks *square* pada sebelah kiri adalah Matrix Vandermonde $(V_n)$ dengan elemen pada $(k, j)$ adalah $\omega_n^{kj}$. Untuk mendapatkan inverse FFT, kita melakukan manipulasi aljabar matriks pada *(14)* sehingga diperoleh:

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & \cdots & w_n^{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

*( 16 )*

Hasil invers matriks Vandermonde tersebut dapat dinyatakan sebagai berikut:

$$\begin{pmatrix} w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & \cdots & w_n^{-2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{pmatrix}$$

*( 17 )*

Berdasarkan matriks tersebut, diperoleh formula inverse FFT;

$$a_k = \frac{1}{n}\sum_{j=0}^{n-1} y_j w_n^{-kj}$$

Perhatikan bahwa, formula tersebut mirip dengan formula FFT:

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{kj}$$

Perbedaan utama dari kedua formula tersebut adalah penukaran $a$ dan $y$ penukaran $\omega_n^k$ dengan $\omega_n^{-k}$, dan pembagian hasil penjumlahan dengan $n$. Namun, secara umum, proses perhitungan FFT dan Inverse FFT tidak jauh berbeda.

### *K. Analisis Kompleksitas Fast Fourier Transform*

Sekarang kita akan mendiskusikan mengenai kompleksitas waktu dari algoritma FFT. Perhatikan bahwa, polinomial $P_1$ dan $P_2$ hanya memiliki setengah dari jumlah koefisien pada polinomial $P$. Jika perhitungan $DFT(P)$ dilakukan dalam waktu linear menggunakan $DFT(P_1)$ dan $DFT(P_2)$, kita dapat memperoleh relasi rekurens sebagai berikut:

$$T_{DFT}(N) = 2T_{DFT}\left(\frac{N}{2}\right) + T(N)$$

Dengan menggunakan Teorema Master yang tidak akan dibahas disini, kita memperoleh kompleksitas waktu $O(Nlog(N))$.

### *L. Number Theoretic Transform (NTT)*

Pada *Number Theoretic Transform* (NTT), selain mengalikan dua buah polinomial dalam waktu $O(Nlog(N))$, kita juga ingin melakukan modulo pada koefisien hasil perkalian suku banyak terhadap suatu bilangan prima *p*. DFT, yang sudah kita bahas sebelumnya, menggunakan bilangan kompleks dan akar satuan ke-*n* dalam proses kalkulasinya. Di sisi lain, perhitungan NTT didasarkan pada bilangan bulat.

Pada dasarnya, DFT biasa dan NTT tidak berbeda jauh. Perbedaan yang paling utama antara DFT normal dan NTT terletak pada definisi akar satuan ke-*n* pada aritmatika modular. Pada NTT, akar satuan ke-*n* didefinisikan sebagai:

$$(\omega_n)^n \equiv 1 \ (mod\ p)$$
$$(\omega_n)^k \not\equiv 1 \ (mod\ p), \qquad 1 \le k < n$$

*( 18 )*

Akar-akar satuan lainnya dapat diperoleh dengan menghitung pangkat dari akar $\omega_n$.

Untuk melakukan algoritma FFT, kita memerlukan akar satuan untuk sembarang $n$, yang merupakan perpangkatan dari 2. Perhatikan bahwa,

$$(w_n^2)^m \equiv w_n^n \equiv 1 \ (mod\ p), \qquad m = \frac{n}{2}$$
$$(w_n^2)^k \equiv w_n^{2k} \not\equiv 1 \ (mod\ p), \qquad 1 \le k < m$$

*( 19 )*

Jadi, jika $\omega_n$ adalah akar satuan ke-n, maka $\omega_n^2$ adalah akar satuan ke-$n/2$. Akibatnya, untuk semua perpangkatan dua yang lebih kecil, terdapat akar satuan yang dapat dihitung menggunakan $\omega_n$. Untuk melakukan inverse DFT, kita memerlukan $w_n^{-1}$. Tinjau bahwa untuk setiap modulus prima, inverse modulo pasti ada sehingga semua sifat dari bilangan kompleks yang dibutuhkan untuk perhitungan FFT, juga ada pada aritmatika modular.

Pada makalah ini, kita menggunakan 998244353 sebagai modulus yang merupakan sebuah bilangan prima dalam bentuk $p = c2^k + 1 = 119 \cdot 2^{23} + 1$. Dapat dibuktikan bahwa $g^c \ mod\ 998244353$ adalah akar satuan ke-$2^k$ dengan $g$ adalah akar primitif dari $p$. Dengan melakukan kalkulasi akar primitif dari $p$ yang tidak akan dijelaskan disini, didapatkan nilai $g = 5$. Akibatnya, dapat diperoleh:

$$g^c \ mod\ 998244353 = 15311432$$

*( 20 )*

dan inverse dari akar satuan tersebut adalah:

$$g^{-c} \ mod\ 998244353 = 469870224$$

*( 21 )*

## III. Analisis dan Implementasi

### *A. Solusi Sederhana*

Pendekatan paling intuitif adalah dengan menghitung semua koefisien binomial dari $(1+x)^n$ kemudian menyimpannya pada sebuah *array* atau *vector* dengan ukuran $2^k$.

Secara umum, langkah-langkah dari algoritma ini adalah sebagai berikut.

1. Lakukan *preprocessing* untuk menghitung hasil faktorial hingga $n$ dan simpan di sebuah *array*. Faktorial-faktorial ini akan digunakan pad saat proses kalkulasi koefisien binomial dari $(1+x)^n$.
2. Untuk setiap $i \in [0, n]$, lakukan perhitungan koefisien suku tersebut sesuai formula berikut:
$$\binom{n}{i} = \frac{n!}{n!\,(n-i)!}$$
Karena kita sedang bekerja dengan bilangan bulat dan modulo $p$, lakukan inverse modulo pada ekspresi $n!\,(n-i)!$. Selanjutnya, simpan hasil tersebut pada sebuah *array* dengan ukuran $2^k$ pada indeks $i \ mod\ 2^k$.

Berikut adalah implementasi dari algoritma ini dalam C++.

naivesolution.cpp

```cpp
#include <bits/stdc++.h>
#define int long long int
using namespace std;

// Constant
const int MOD = 998244353;

// Array to save factorial results from preprocessing
vector <int> fact;

// Function that returns a^b mod m
int binpow(int a, int b, int m)
{
    int res = 1;
    while (b)
    {
        if (b & 1)
            res = (res * a) % m;
        a = (a * a) % m;
        b /= 2;
    }
    return res;
}
```

**Gambar 2**. Program untuk solusi sederhana dalam C++ (bagian 1)

```cpp
// Preprocessing procedure
void prep(int n)
{
    fact.resize(n + 1, 0);
    fact[0] = 1;
    for (int i = 1; i <= n; i++)
    {
        fact[i] = (i * fact[i - 1]) % MOD;
    }
}

int32_t main()
{
    auto start = std::chrono::high_resolution_clock::now();
    int n, k, kPow;
    cin >> k >> n; // Read n and k
    kPow = 1 << k;
    prep(n);
    vector <int> a(kPow, 0);

    // Calculate coefficients for all terms
    for (int i = 0; i <= n; i++)
    {
        int denominator = (fact[i] * fact[n - i]) % MOD;
        int numerator = fact[n];
        a[i % kPow] = (a[i % kPow] + (numerator * binpow(denominator, MOD - 2, MOD)) % MOD) % MOD;
    }

    // Show answer
    for (int i = 0; i < kPow; i++)
    {
        cout << a[i] << ' ';
    }
    cout << '\n';
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    cout << "Execution time: " << duration.count() << " seconds." << std::endl;
    return 0;
}
```

**Gambar 3**. Program untuk solusi sederhana dalam C++ (bagian 2)

Tinjau bahwa saat *preprocessing*, terdapat operasi sebanyak $n$ kali sehingga waktu yang dibutuhkan adalah $n$. Pada perhitungan koefisien binomial, program melakukan iterasi sebanyak $n + 1$ kali. Untuk setiap iterasi, terjadi pemanggilan fungsi *binpow* yang memakan waktu $\log 998244351$. Jadi, kompleksitas waktu total adalah

$$T(n) = 2n + (n+1)\log 998244351 \approx 32n + 30$$

Perhatikan bahwa,

$$T(n) = 32n + 30 \leq 32n + n \leq 33n$$

dengan $n \geq 30$. Akibatnya, kompleksitas waktu asimptotiknya adalah $T(n) = O(n)$.

Algoritma ini bekerja dengan cukup baik pada $n \leq 10^7$. Namun, pada nilai $n$ yang lebih besar, algoritma ini menjadi sangat lambat. Selain itu, keterbatasan memori juga menjadi *bottleneck* karena kita menyimpan semua nilai faktorial hingga $n$. Akibatnya, kita memerlukan pendekatan lain yang lebih cepat dan hemat secara memori.

### *B. Solusi Menggunakan FFT*

Salah satu sifat menarik pada FFT adalah derajat suatu polinomial bersifat sirkular. Dengan kata lain,

$$x^a \cdot x^b = x^{(a+b)\ mod\ L}$$

Pada persamaan tersebut, $L$ adalah ukuran dari suatu FFT dengan $L = 2^k$.

Pada solusi ini, kita menggunakan sifat FFT yang sirkular untuk melakukan kalkulasi koefisien setiap suku $(1 + x)^n$. Jadi, meskipun $n$ sangat besar, derajat setiap suku pada $(1 + x)^n$ diberikan modulo $2^k$. Pada akhirnya, kita hanya akan memerlukan *array* dengan kapasitas $2^k$.

Secara umum, langkah-langkah dari algoritma ini adalah sebagai berikut.

1. Definisikan polinomial $p(x) = 1 + x$ dan simpan pada suatu *array* dengan ukuran $2^k$. Hal ini dilakukan untuk mengatur ukuran FFT menjadi $2^k$ agar sifat sirkular berlaku.
2. Lakukan perkalian polinom $DFT(p)$ dengan dirinya sendiri sebanyak $n$ kali. Dengan kata lain, kita akan menghitung $(1 + x)^n$. Untuk melakukan hal ini, kita hanya perlu memangkatkan setiap elemen pada vektor $DFT(p)$.
3. Lakukan inverse FFT pada $DFT(p^n)$ sehingga diperoleh vektor koefisien yang merupakan barisan yang kita inginkan.

Berikut adalah implementasi algoritma tersebut menggunakan bahasa C++.

fft.cpp

```cpp
#include <bits/stdc++.h>
#include <chrono>
#define int long long int
using namespace std;

// constants
const int MAXN = 5e5;
const int MOD = 998244353;
const int root = 15311432;
const int root_1 = 469870224;
const int root_pw = 1 << 23;

// function that returns a^b mod m
int binpow(int a, int b, int m)
{
    int res = 1;
    while (b)
    {
        if (b & 1)
            res = (res * a) % m;
        a = (a * a) % m;
        b /= 2;
    }
    return res;
}

// recursive FFT
void fft (vector <int> &a, bool invert)
{
    int n = a.size();
    if (n == 1)
        return;
    vector <int> a0(n / 2), a1(n / 2);
    for (int i = 0; 2 * i < n; i++)
    {
        a0[i] = a[2 * i];
        a1[i] = a[2 * i + 1];
    }
    // do recursive call on a0 and a1
    fft(a0, invert);
    fft(a1, invert);

    int w = 1, wn = invert? root_1 : root;

    for (int i = n; i < root_pw; i <<= 1)
        wn = (wn * wn) % MOD;

    for (int i = 0; 2 * i < n; i++)
    {
        int u = a0[i], v = (w * a1[i]) % MOD;
        a[i] = (u + v) % MOD;
        a[i + n / 2] = (u - v + MOD) % MOD;

        if (invert)
        {
            a[i] = (a[i] * 499122177) % MOD; // a[i] /= 2 but in modular arithmetic
            a[i + n / 2] = (a[i + n / 2] * 499122177) % MOD; // a[i + n / 2] /= 2 but in modular arithmetic
        }
        w = (1LL * w * wn % MOD);
    }
}
```

**Gambar 4**. Program menggunakan FFT dalam bahasa C++ (bagian FFT)

```cpp
int32_t main()
{
    auto start = std::chrono::high_resolution_clock::now();
    vector <int> a;
    int n, k;
    cin >> k >> n; // input n and k
    int l = 1 << k;
    a.resize(l, 0); // set array size to 2^k
    a[0] = 1;
    a[1] = 1;
    // do fft
    fft(a, false);
    // polynomial multiplication of (1 + x)^n
    for (int i = 0; i < l; i++)
        a[i] = binpow(a[i], n, MOD);
    // inverse fft
    fft(a, true);
    for (int i = 0; i < l; i++)
    {
        cout << a[i] << ' ';
    }
    cout << endl;
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    cout << "Execution time: " << duration.count() << " miliseconds." << std::endl;
    return 0;
}
```

**Gambar 5**. Program menggunakan FFT dalam bahasa C++ (bagian *main*)

Seperti yang sudah dibahas sebelumnya, proses FFT dan *inverse* FFT memiliki kompleksitas waktu sebesar $O(N \log N)$ dengan $N$ adalah ukuran FFT. Dalam hal ini, ukuran FFT tersebut adalah $2^k$ sehingga kompleksitas waktu untuk melakukan FFT dan inverse FFT adalah

$$O(k2^k \log(2)) = O(k2^k).$$

Kompleksitas waktu asimptotik dari proses perkalian polinomial adalah $O(2^k \log 998244353) = O(2^k)$. Jadi, kompleksitas waktu total adalah

$$O(\max(2^k, k2^k)) = O(k2^k)$$

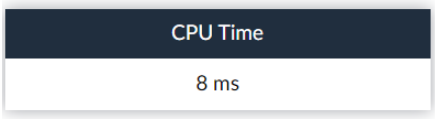
Dengan $k \leq 17$ sehingga proses kalkulasi dengan FFT jauh lebih cepat dan hemat memori dibandingkan algoritma sederhana.

## IV. Studi Kasus

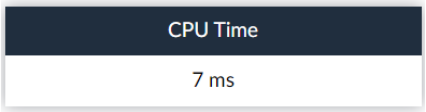
Pada bagian ini kita akan mencoba membandingkan kecepatan proses kalkulasi dari dua algoritma yang sudah kita bahas, yaitu algoritma sederhana dan algoritma menggunakan FFT.

### *A. Uji Kasus 1*

Pada kasus uji ini, kita akan menggunakan kasus uji pada $k$ dan $n$ yang kecil yaitu $k = 3$ dan $n = 10$. Berikut adalah perbandingan kedua buah algoritma.




**Gambar 6**. Hasil uji kasus 1 menggunakan algoritma sederhana

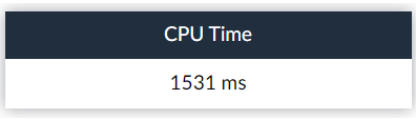



**Gambar 7**. Hasil uji kasus 1 menggunakan algoritma FFT

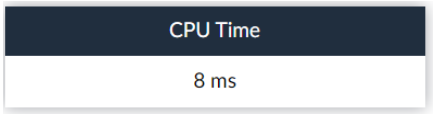
Perhatikan bahwa, pada uji kasus ini waktu komputasi sangat cepat dan hampir tidak ada perbedaan di antara kedua algoritma.

### *B. Uji Kasus 2*

Pada kasus uji ini, kita akan menggunakan kasus uji pada $k$ yang tidak terlalu besar dan $n$ yang cukup besar, yaitu $k = 10$ dan $n = 10000000$. Berikut adalah perbandingan kedua buah algoritma tersebut.




**Gambar 8**. Hasil uji kasus 2 menggunakan algoritma sederhana

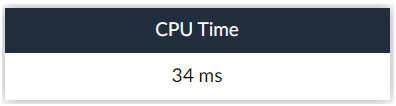



**Gambar 9**. Hasil uji kasus 2 menggunakan algoritma FFT

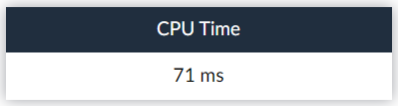
Dapat dilihat bahwa ada perbedaan waktu yang sangat signifikan antara algoritma sederhana dan algoritma dengan FFT. Hal ini karena kompleksitas waktu algoritma sederhana bergantung pada $n$, sedangkan FFT bergantung pada $k$.

### *C. Uji Kasus 3*

Pada uji kasus ini, kita akan menggunakan $k$ yang besar, dan $n$ yang tidak terlalu besar, yaitu $k = 17$ dan $n = 100000$. Berikut adalah hasil perbandingan kedua algoritma tersebut.




**Gambar 10**. Hasil uji kasus 3 menggunakan algoritma sederhana

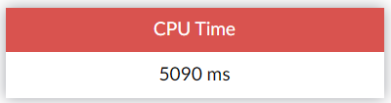



**Gambar 11**. Hasil uji kasus 3 menggunakan algoritma FFT

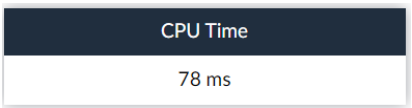
Perhatikan bahwa, algoritma FFT memakan waktu yang sedikit lebih lama dibandingkan algoritma sederhana. Hal ini karena, algoritma FFT bergantung pada nilai $k$ yang besar, sedangkan algoritma sederhana bergantung pada nilai $n$ yang relatif lebih kecil.

### *D. Uji Kasus 4*

Pada kasus uji ini, kita akan menggunakan kasus uji pada $k$ dan $n$ yang besar, yaitu $k = 17$ dan $n = 100000000$. Berikut adalah hasil perbandingan kedua buah algoritma.




**Gambar 12**. Hasil uji kasus 4 menggunakan algoritma sederhana




**Gambar 13**. Hasil uji kasus 4 menggunakan algoritma FFT

Berdasarkan perbandingan tersebut dapat dilihat bahwa terdapat perbedaan waktu yang sangat signifikan antara kedua algoritma. Hal ini sesuai dengan analisis pada bagian sebelumnya.

## V. Kesimpulan

Algoritma Fast Fourier Transform (FFT) memiliki banyak aplikasi di bidang informatika. Salah satu contoh penerapan algoritma FFT adalah untuk melakukan kalkulasi barisan modular dari koefisien ekspansi binomial. Algoritma ini

memungkinkan proses kalkulasi yang lebih cepat dibandingkan pendekatan naif atau sederhana. Selain itu, penggunaan memori oleh algoritma FFT juga lebih efisien. Hal ini sudah terbukti berdasarkan *runtime* algoritma FFT yang secara umum jauh lebih baik dibandingkan pendekatan biasa.

## VI. Ucapan Terima Kasih

Penulis mengucapkan rasa syukur kepada Tuhan Yang Maha Esa karena berkat rahmat-Nya, penulis bisa menyelesaikan makalah "Implementasi Fast Fourier Transform Dalam Kalkulasi Barisan Modular Dari Koefisien Ekspansi Binomial" dengan baik. Penulis juga ingin mengucapkan terima kasih kepada dosen mata kuliah Matematika Diskrit, Dr. Nur Ulfa Maulidevi, S. T, M.Sc., Dr. Ir. Rinaldi Munir, M. T., dan Dr. Fariska Zakhralativa Ruskanda, S.T. yang telah membimbing penulis selama mengikuti pembelajaran pada mata kuliah ini. Tidak lupa, penulis ingin mengucapkan terima kasih kepada seluruh sumber yang dijadikan referensi pada makalah ini.

## Referensi

[1] R. Munir, "Kombinatorial (Bagian 1)." 2020. Diakses: 11 Desember 2023. [Daring]. Tersedia pada: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/17-Kombinatorial-Bagian1-2023.pdf

[2] R. Munir, "Kombinatorial (Bagian 2)." Diakses: 11 Desember 2023. [Daring]. Tersedia pada: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/18-Kombinatorial-Bagian2-2023.pdf

[3] "Complex Numbers | Brilliant Math & Science Wiki." Diakses: 11 Desember 2023. [Daring]. Tersedia pada: https://brilliant.org/wiki/complex-numbers/

[4] "Roots of Unity | Brilliant Math & Science Wiki." Diakses: 11 Desember 2023. [Daring]. Tersedia pada: https://brilliant.org/wiki/roots-of-unity/

[5] "Fast Fourier transform - Algorithms for Competitive Programming." Diakses: 11 Desember 2023. [Daring]. Tersedia pada: https://cp-algorithms.com/algebra/fft.html

[6] "Primitive Root - Algorithms for Competitive Programming." Diakses: 11 Desember 2023. [Daring]. Tersedia pada: https://cp-algorithms.com/algebra/primitive-root.html

[7] "Euler's Totient Function | Brilliant Math & Science Wiki." Diakses: 11 Desember 2023. [Daring]. Tersedia pada: https://brilliant.org/wiki/eulers-totient-function/

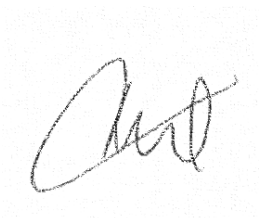
## Tautan Kode Program

https://github.com/ninoaddict/Program-Matematika-Diskrit

## Peryataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2023



Adril Putra Merin 13522068