Penerapan Divide And Conquer Dalam Mengalikan Polinomial

Jauhar Arifin—13515049
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13515049@std.stei.itb.ac.id

***Abstrak*—Penyelesaian permasalah perkalian pada fungsi polynomial merupakan masalah yang sering dijumpai dalam bidang matematika. Sekarang, kebutuhan untuk melakukan perkalian pada polynomial semakin meningkat. Oleh karena itu dibutuhkan algoritma yang cukup cepat yang efektif untuk menyelesaikan masalah ini. Salah satu pendekatan untuk menyelesaikan masalah ini adalah dengan menggunakan algoritma berbasis *divide and conquer*.**

Kata kunci—divide and conquer; polynomial; multiplication; big integer; fast fourier transform; karatsuba

I. PENDAHULUAN

Polinomial merupakan fungsi yang sangat dibutuhkan dalam dunia matematika dalam memodelkan sesuatu. Terkadang, polinomial digunakan dalam mengaproksimasi suatu fungsi yang tidak diketahui. Sering kali fungsi tersebut perlu untuk dikalikan dengan fungsi polinomial lain. Fungsi polinomial bisa dikatakan merupakan bentuk fungsi yang sederhana karena mudah dihitung nilainya, diintegral dan diturunkan. Oleh karena itu, hasil perkalian dari dua buah fungsi polinomial hasil aproksimasi dari suatu fungsi lain juga dapat dengan mudah dihitung, diintegral dan diturunkan karena hasil perkalian dari fungsi polinomial juga merupakan fungsi polinomial.

Perkalian polinomial juga berguna dalam bidang matematika abstrak. Polinomial pada matematika abstrak digunakan sebagai salah satu bentuk dari field. Aplikasi perkalian polinomial dalam hal ini berhubungan dengan kriptografi. Berbagai algoritma kriptografi didasari dengan perkalian polinomial. Tidak hanya perkalian saja, pada bidang kriptografi, operasi pada polinomial juga melibatkan penjumlahan dan modulo.

Aplikasi polinomial yang paling sering digunakan adalah untuk merepresentasikan suatu bilangan. Semua bilangan umumnya adalah polinomial dengan suatu nilai variabel tertentu. Bilangan desimal merupakan polinomial dengan nilai variabel sepuluh sedangkan bilangan biner merupakan polinomial dengan nilai variabel dua. Perkalian antara dua buah bilangan sangat sering dilakukan, oleh karena itu pada dasarnya perkalian polinomial juga sering dilakukan. Hal tersebut didasari oleh fakta bahwa semua bilangan umumnya merupakan suatu polinomial.

Dengan menggunakan definisi polinomial dan perkalian polinomial, algoritma perkalian polinomial dapat diciptakan berdasarkan definisi yang ada. Meskipun begitu, algoritma berdasarkan definisi tersebut masih dinilai terlalu kompleks terhadap waktu atau dengan kata lain masih terlalu lambat. Karena kebutuhan akan polinomial yang semakin tinggi tentunya diperlukan algoritma dengan kompleksitas waktu yang lebih sederhana.

Dengan menggunakan pendekatan *divide and conquer*, permasalahan perkalian polinomial dapat disederhanakan sehingga kompliksitas waktunya lebih sederhana. Terdapat beberapa algoritma yang memanfaatkan pendekatan *divide and conquer* untuk menyelesaikan masalah ini. Dalam makalah ini, pendekatan *divide and conquer* akan diaplikasikan dalam menyelesaikan permasalahan perkalian polinomial ini.

II. DASAR TEORI

A. Polinomial

Kita mendefinisikan suatu bentuk monomial dengan variabel x sebagai ekspresi berbentuk :

$$ax^n$$

Dimana a merupakan bilangan riil dan n merupakan bilangan bulat tak negative. Dalam bentuk tersebut a disebut sebagia koefisien dari monomial ax^n, sedangkan n disebut sebagai exponent dari monomial ax^n[3].

Suatu polinomial dengan variabel x merupakan bentuk penjumlahan dari berhingga monomial yang berbeda. Suatu polinomial biasanya disimbolkan dengan $f(x), g(x), h(x)$,dll. Secara definisi, polinomial dapat dituliskan dalam bentuk ekspresi sebagai berikut :

$$a_0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n$$

Polinomial tersebut disebut berderajat n. Derajat dari polinomial merupakan nilai eksponen tertinggi dari seluruh suku polinomial yang nilai koefisiennya bukan nol[4]. Secara definisi, derajat dari polinomial $f(x)$ dapat dituliskan sebagai $\deg f(x)$.

B. Perkalian Polinomial

Beberapa operasi aritmatika dapat dilakukan pada polinomial seperti penjumlahan, pengurangan, perkalian dan pembagian. Operasi penjumlahan dan perkalian cukup mudah untuk dilakukan yaitu dengan menjumlahkan setiap suku polinomial yang memiliki nilai eksponen yang sama. Proses

tersebut dapat dilakukan dalam kompleksitas waktu $O(N)$ untuk menjumlahkan dua buah polinomial berderajat N. Operasi penjumlahan dan pengurangan memang cukup mudah dilakukan, meskipun begitu terdapat operasi yang sulit dilakukan yaitu perkalian.

Fakta menarik dari operasi pada polinomial adalah bahwa hasil dari penjumlahan, pengurangan dan perkalian dari dua buah polinomial merupakan sebuah polinomial. Untuk operasi penjumlahan dua buah polinomial yang berderajat N dan M, polinomial yang dihasilkan akan berderajat $\max\{N, M\}$. Sedangkan untuk operasi perkalian, dua buah polinomial dengan derajat N dan M akan memberikan hasil kali berupa polinomial dengan derajat $N + M$.

Perkalian polinomial lebih sulit dilakukan dibandingkan dengan penjumlahannya. Tidak seperti penjumlahan, untuk menghitung koefisien sebuah suku pada hasil perkalian dua buah polinomial berderajat N dibutuhkan untuk menghitung hasil perkalian dari N buah koefisien dari setiap polinomial. Secara definisi hasil perkalian dua buah polinomial $f(x)$ dan $g(x)$ berderajat n sebagai berikut :

$$f(x) = a_0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n$$
$$g(x) = b_0 + b_1x^1 + b_2x^2 + \cdots + b_nx^n$$

adalah

$$f(x)g(x) = h(x) = c_0 + c_1x^1 + c_2x^2 + \cdots + c_{2n}x^{2n}$$

dengan

$$c_i = \sum_{\max\{0,i-n\} \le k \le \min\{i,n\}} a_k b_{i-k}$$

Dari definisi diatas dapat disimpulkan, untuk mengalikan polinomial $f(x)$ dan $g(x)$, setiap suku a_i perlu dikalikan dengan setiap suku b_i. Oleh karena itu untuk setiap perkalian dua buah polinomial dengan derajat N, paling banyak terdapat N^2 perkalian yang perlu dihitung.

C. Discrete Fourier Transform

Discrete Fourier Transform atau biasa disingkat DFT merupakan suatu pemetaan dari vektor a menjadi $\hat{a}$ sebagai berikut :

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \rightarrow \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_{n-1} \end{pmatrix}$$

dimana,

$$\hat{a}_i = \sum_{k=0}^{n-1} a_k \omega_n^{ik}$$

dengan $\omega_n = e^{i\left(\frac{2\pi}{n}\right)}$ = akar kompleks ke-n dari satu. Pada dasarnya DFT melakukan evaluasi nilai polinomial dari suatu polinomial pada n buah nilai yaitu $\omega_n^0, \omega_n^1, \cdots, \omega_n^{n-1}$. Hasil evaluasi tersebut dapat dituliskan dalam bentuk perkalian matriks sebagai berikut.

$$\hat{a} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{pmatrix} a$$

Matriks yang digunakan untuk menentukan nilai $\hat{a}$ diatas dinamakan matriks Vandermonde dan biasananya ditulis menggunakan simbel V.

Hal menarik dari DFT adalah fakta bahwa dengan mengetahui nilai $\hat{a}$, nilai dari a juga dapat ditentukan dengan mudah. Hal ini berarti dengan mengetahui nilai suatu polinomial berderajat n di n titik tertentu, bentuk dari polinomial tersebut dapat ditentukan dengan mudah.

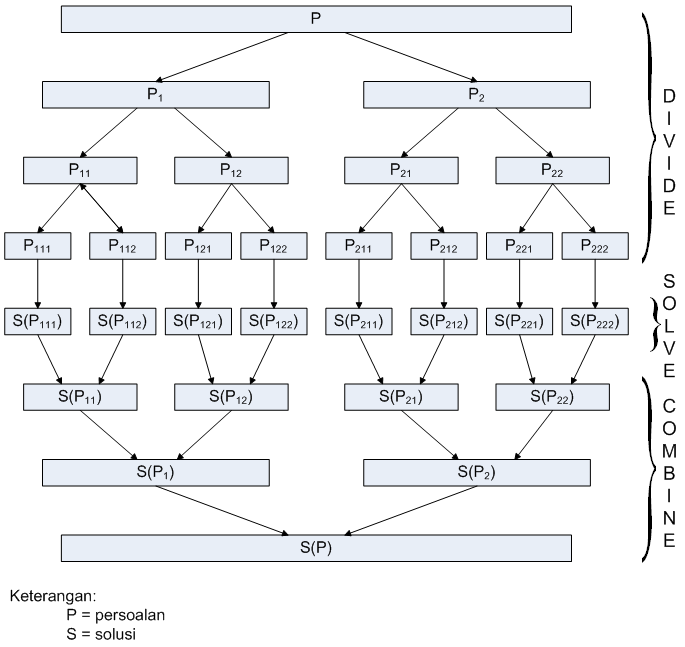
Untuk menentukan nilai a dengan informasi $\hat{a}$, sebuah matriks inverse dari V dapat dibentuk, yaitu matriks V^{-1} :

$$V^{-1} = \frac{1}{n}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \cdots & \omega_n^{-(n-1)^2} \end{pmatrix}$$

Kemudian dengan menghitung nilai dari $V^{-1}\hat{a}$, nilai a dapat ditentukan.

D. Divide And Conquer

Divide and Conquer merupakan salah satu bentuk pendekatan dalam menyelesaikan suatu permasalahan. Pendekatan *divide and conquer* bekerja dengan cara memecah permasalahan menjadi *submasalah* yang sama seperti masalah awal. Setiap *submasalah* diselesaikan secara terpisah dengan cara rekursif. Solusi dari setiap *submasalah* akan digabung menjadi solusi masalah utama[5].



Gambar 1. Ilustrasi Pendekatan *Divide And Conquer*

Tidak semua permasalah dapat menjadi lebih baik jika diselesaikan dengan pendekatan *divide and conquer*. Beberapa

persoalan bisa jadi malah bertambah kompleks jika diselesaikan dengan menggunakan *divide and conquer* dibandingkan dengan cara *bruteforce*. Oleh karena itu, pendekatan *divide and conquer* tidak dapat langsung digunakan begitu saja melainkan harus disertai dengan cara memecah persoalan sehingga dapat menjadi lebih sederhana.

Dengan menggunakan pendekatan *divide and conquer*, kompleksitas waktu dari suatu algoritma dapat dihitung dengan menggunakan Teorema Master.

E. Teorema Master

Dalam menghitung kompleksitas waktu suatu algoritma yang berjalan dengan konsep *divide and conquer*, teorema master dapat digunakan untuk mempermudah perhitungan. Teorema master memberikan hubungan antara bentuk rekurens suatu kompleksitas dengan kompleksitas waktu asimptotik. Dengan menggunakan teorema master, hanya dengan mengetahui bentuk rekurens dari $T(N)$ saja, nilai Big-O dapat ditentukan dengan mudah.

Teorema master mengacu pada bentuk rekurens :

$$T(N) = aT\left(\frac{N}{b}\right) + cn^d$$

Nilai a menyatakan banyaknya *submasalah* hasil dari pemecahan masalah utama. N merupakan ukuran dari masalah utama. Oleh karena itu $\frac{N}{b}$ merupakan ukuran dari *submasalah* hasil pemecahan masalah utama. cn^d menyatakan komputasi yang terjadi di luar pemanggilan fungsi secara rekursif, contohnya seperti : komplekitas pemecahan masalah utama menjadi *submasalah* dan menggabungkan solusi dari *submasalah* menjadi solusi sebenarnya.

Nilai Big-O yang dihasilkan oleh teorema master bergantung dari nilai a, b dan $f(n)$ pada ekspresi rekurens kompleksitas waktu. Terdapat tiga kasus dalam menentukan nilai Big-O, yaitu[1]:

1. Nilai $T(N) = O(n^d)$, jika $a < b^d$
2. Nilai $T(N) = O(n^d \log n)$, jika $a = b^d$
3. Nilai $T(N) = O\left(n^{\log_b a}\right)$, jika $a > b^d$

III. Penyelesaian Perkalian Polinomial Dengan Divide And Conquer

Untuk menyelesaikan permasalahan perkalian polinomial dengan menggunakan pendekatan *divide and conquer*, beberapa algoritma dapat digunakan. Dalam makalah ini, akan dipaparkan dua buah algoritma untuk mengalikan dua buah polinomial dengan menggunakan pendekatan *divide and conquer*.

Untuk lebih memudahkan proses pengalian dua buah polinomial, suatu polinomial f dapat diibaratkan sebagai sebuah $array$ `f` dimanana nilai dari `f[i]` adalah koefisien dari polinomial f yang memiliki variabel x^i. Dua buah polinomial f dan g dengan derajat N dan M dapat direpresentasikan menjadi array `f` dan `g` dengan panjang $N + 1$ dan $M + 1$. Untuk memudahkan, setiap perkalian antara dua buah polinomial f dan g, representasi *array* dari polinomial yang memiliki derajat lebih kecil dapat diperlebar sehingga dua buah polinomial memilki representasi *array* dengan panjang yang sama.

Pada dasarnya, setiap polinomial f dapat direpresentasikan menjadi *array* `f` dengan mengisi nilai a_i pada suku $a_i x^i$ pada polinomial f kedalam array `f` pada inde ke-i. Sedangkan untuk memperlebar ukuran array f dapat dilakukan dengan mengisikan nilai nol pada array `f` yang kurang. Misalnya untuk merepresentasikan polinomial $f(x)$ sebagai berikut :

$$f(x) = 1 + 3x + 2x^2 + x^4 + 2x^5$$

array `f` dengan panjang array 6 dapat diciptakan dengan nilai sebagai berikut :

```
f[0] = 1
f[1] = 3
f[2] = 2
f[3] = 0
f[4] = 1
f[5] = 2
```

Sedangkan untuk memperlebar array tersebut sehingga memiliki panjang misalnya delapan, array tersebut dapat diperlebar menjadi :

```
f[0] = 1
f[1] = 3
f[2] = 2
f[3] = 0
f[4] = 1
f[5] = 2
f[6] = 0
f[7] = 0
```

A. Algoritma Naif

Untuk mengalikan dua buah polinomial $f(x)$ dan $g(x)$, sesuai definisi, setiap suku pada polinomial f perlu dikalikan dengan setiap suku pada polinomial g. Hal ini dapat dilakukan menggunakan algoritma *traversal* pada *array*. Dengan merepresentasikan polinomial menggunakan *array* satu dimensi, algoritma naif dapat dituliskan sebagai berikut :

```
1. def naive_algorithm(f, g):
2.   hasil = [0.0] * (len(f) + len(g) - 1)
3.   for i in range(0, len(f)):
4.     for j in range(0, len(g)):
5.       hasil[i+j] += f[i] * g[j]
6.   return hasil
```

Listing 1. Implementasi Algoritma Naif

Dalam algoritma tersebut, dapat diperhatikan bahwa untuk setiap suku $a_i x^i$ pada polinomial f dikalian dengan setiap suku $b_j x^j$ pada polinomial g menjadi kontribusi dari sebuah suku pada polinomial hasil dengan variabel x^{i+j}. Oleh karena itu terdapat NM buah operasi perkalian dengan menggunakan algoritma naif. Algoritma ini bekerja dengan cara mengikuti definisi matematis tentang perkalian dua buah polinomial.

Dengan menggunakan algoritma ini, jika ukuran array polinomal f dan g adalah N maka terdapat N^2 buah operasi perkalian dalam menghitung hasil perkalian atara f dan g. Oleh karena itu, kompleksitas waktu algoritma ini adalah $O(N^2)$. Nilai kompleksitas dari algoritma ini juga dapat dilihat dari adanya dua buah *loop* yang bersarang pada baris ke-3 dan ke-4

pada Listing 1. Kedua buah *loop* tersebut memiliki N buah proses yang perlu dieksekusi, oleh karena terdapat $N \times N$ buah perintah yang perlu dieksekusi, sehingga kompleksitas waktunya menjadi $O(N^2)$.

Untuk nilai N yang kecil algoritma ini masih cocok untuk digunakan. Misalnya untuk nilai $N = 10$, algoritma ini masih tergolong cepat untuk dieksekusi. Nyatanya dengan nilai N yang masih dibawah 1000, dengan komputer yang ada sekarang, algoritma ini dapat berjalan kurang dari satu detik. Sedangkan dengan nilai N dibawah 100 algoritma ini dapat berjalan kurang dari 0.3 detik.

Meskipun algoritma ini tidak cukup baik secara kompleksitas waktu, algoritma ini memiliki kompleksitas memori yang sangat sederhana yaitu $O(N)$. Karena memori yang diperlukan oleh algoritma ini hanyalan memori untuk menyimpan *array* dari polinomial f, g dan polinomial hasil, maka diperlukan kompleksitas memori sebesar $O(N + N + 2N) = O(N)$.

Algoritma ini tidak membutuhkan memori yang besar untuk melakukan operasinya sehingga untuk kebanyakan kasus perkalian dengan nilai N yang kecil, algoritma ini lebih sering digunakan dibandingkan dengan algoritma yang menggunakan pendekatan *divide and conquer* karena umumnya algoritma yang berbasis *divide and conquer* dikerjakan secara rekurisf sehingga diperlukan ruang tambahan untuk melakukan penyimpanan *stack* pemanggilan fungsi. Contoh penggunaan algoritma ini adalah pada proses perkalian bilangan dengan tipe `int, long, short, char, long long` pada bahasa pemrograman C.

B. Algoritma Karatsuba

Algoritma Karatsuba merupakan algoritma *fast multiplication* berbasis *divide and conquer* yang diciptakan oleh Anatoly Karatsuba pada tahun 1960. Algoritma Karatsuba dapat memcah permasalah perkalian dua buah polinomial berderajat N menjadi tiga buah perkalian polinomial berderajat $N/2$. Berdasarkan Teorema Master, dapat dibuktikan bahwa kompleksitas waktu dari Algoritma Karatsuba adalah $O(N^{\log_2 3})$.

Pada dasarnya setiap polinomial $f(x)$ dengan derajat N dapat dituliskan sebagai penjumlahan dua buah polinomial berderajat $N/2$. Misalkan $f(x)$ adalah sebuah polinomial berderajat N dengan N adalah bilangan ganjil positif sebagi berikut :

$$f(x) = a_0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n$$

Bentuk polinomial f tersebut dapat diubah menjadi bentuk penjumlahan antara dua buah polinomial lain yang memiliki derajat $(N-1)/2$ sebagi berikut

$$f(x) = \left(a_0 + a_1x^1 + \cdots + a_{(n-1)/2}x^{(n-1)/2}\right) + \left(a_{\frac{n-1}{2}+1} + a_{\frac{n-1}{2}+2}x^1 \cdots + a_nx^{\frac{n-1}{2}}\right)x^{\frac{n+1}{2}}$$

Sehingga polinomial f dapat ditulis sebagai berikut :

$$f(x) = g(x) + h(x)x^{(n+1)/2}$$

dimana,

$$g(x) = a_0 + a_1x^1 + \cdots + a_{(n-1)/2}x^{(n-1)/2}$$
$$h(x) = a_{(n-1)/2+1} + a_{(n-1)/2+2}x^1 \cdots + a_nx^{(n-1)/2}$$

Contoh pemecahan suatu polinomial adalah sebagai berikut : Misalkan suatu polinomial $f(x)$ terdefisinisi sebagai berikut :

$$f(x) = 1 + 3x + 2x^2 + x^4 + 2x^5$$

Untuk memecah polinoomial tersebut, polinomial f dapat dituliskan menjadi :

$$f(x) = (1 + 3x + 2x^2) + (x + 2x^2)x^3$$

Sehingga f dapat dipecah menjadi dua buah polinomial $g(x)$ dan $h(x)$ sebagai berikut :

$$f(x) = g(x) + h(x)x^3$$

dengan,

$$g(x) = 1 + 3x + 2x^2$$
$$h(x) = x + 2x^2$$

Karena perkalian suatu polinomial $f(x)$ dengan sebuah ekpresi berbentuk x^k pada dasarnya hanyalah menggeser setiap koefisien pada f sebanyak k atau dapat dikatakan meningkatkan nilai eksponen dari setiap suku pada polinomial f sebanyak k, maka proses ini dapat dilakukan dengan mudah. Contohnya hasil perhitungan nilai $h(x)x^3$ pada contoh diatas adalah sebagai berikut :

$$h(x)x^3 = (x + 2x^2)x^3$$
$$= x^4 + 2x^5$$

Pada contoh di atas, nilai eksponen dari setiap suku pada polinomial $h(x)$ bertambah sebanyak tiga yaitu sesuai nilai k pada x^3. Untuk representasinya dalam bentuk *array*, proses menghitung nilai $h(x)x^3$ lebih mudah dilakukan yaitu hanya dengan menggeser posisi pada setiap elemen pada *array*. Untuk contoh kasus yang sama, proses membentuk representasi dari polinomial $h(x)x^3$ adalah sebagai berikut:

h	0	1	2
x	0	1	2

$\times x^3 =$

h	0	0	0	0	1	2
x	0	1	2	3	4	5

Dari ilutstrasi diatas dapat dilihat bahwa proses mengalikan suatu polinomial dengan x^k adalah hanya dengan menggeser setiap elemen pada representasi *array*nya ke kanan sebanyak k kali. Proses ini dapat dilakukan dengan kompleksitas $O(N)$ dengan N menyatakan derajat dari polinomial yang akan dikalikan.

Selain mengalikan polinomial dengan suatu nilai x^k, untuk melakukan pemecahan polinomial diperlukan juga proses untuk memecah polinomial $f(x)$ menjadi $g(x)$ dan $h(x)$. Proses pemecahan polinomial ini sebenarnya mirip dengan proses mengalikan polinomial dengan x^k yaitu hanya dengan menggeser representasi *array*nya saja. Untuk contoh kasus yang sama, proses pemecahan polinomial $f(x) = 1 + 3x + 2x^2 + x^4 + 2x^5$ adalah sebagai berikut :

f	1	3	2	0	1	2
x	0	1	2	3	4	5

menjadi :

g	1	3	2
x	0	1	2

+

h	0	1	2
x	0	1	2

$\times x^3$

Proses tersebut dapat dilakukan dengan kompleksitas waktu $O(N)$. Langkah yang perlu dilakukan adalah dengan membentuk dua buah *array* baru yang bernama `g` dan `h`. Elemen dari `g` merupakan setengah elemen pertama dari `f` sedangkan setengan sisanya menjadi nilai `h` setelah digeser ke kiri sebanyak $\frac{N}{2}$. Untuk selanjutnya setiap polinomial hasil pemecahan polinomial f akan disebut sebagai f_0 dan f_1 dengan $f(x) = f_0(x) + f_1(x)x^{(N+1)/2}$.

```
1.   def reduce(f):
2.     g = [0.0] * ((len(f) + 1) // 2)
3.     h = [0.0] * ((len(f) + 1) // 2)
4.     for i in range(len(f)):
5.       if (i < (len(f)+1)//2):
6.         g[i] = f[i]
7.       else:
8.         h[i-(len(f)+1)//2] = f[i]
9.     return (g,h)
10.
11.  def combine(g,h):
12.    f = []
13.    for x in g:
14.      f.append(x)
15.    for x in h:
16.      f.append(x)
17.    return f
```

Listing 2. Implementasi Pemecahan Dan Penggabungan Polinomial

Untuk mengalikan dua buah polinomial $f(x)$ dan $g(x)$ yang berderajat N, pemecahan dapat dilakukan pada kedua polinomial. Hasil perkalian atara f dan g hasil pemecahan adalah sebagai berikut :

$$\begin{aligned} f(x)g(x) &= (f_0(x) + f_1(x)x^m)\times(g_0(x) + g_1(x)x^m) \\ &= f_0(x)g_0(x) + \big(f_0(x)g_1(x) + f_1(x)g_0(x)\big)x^m \\ &\quad + f_1(x)g_0(x)x^{2m} \\ &= c_0 + c_1x^m + c_2x^{2m} \end{aligned}$$

dimana,

$$m = \frac{N+1}{2}$$
$$c_0 = f_0(x)g_0(x)$$
$$c_1 = f_0(x)g_1(x) + f_1(x)g_0(x)$$
$$c_2 = f_1(x)g_0(x)$$

Sekilas hasil pemecahan tersebut tidak memberikan arti karena jumlah perkalian yang harus dilakukan masih terlalu banyak yaitu sebanyak empat buah perkalian : satu perkalian dari c_0, dua perkalian dari c_1 dan satu perkalian dari c_2. Akan tetapi, Algoritma Karatsuba dapat mereduksi jumlah perkalian ini menjadi tiga buah perkalian saja. Jika diperhatikan nilai dari c_1 dapat dihitung dengan cara sebagai berikut :

$$c_1 = (f_0(x) + f_1(x))(g_0(x) + g_1(x)) - c_2 - c_0$$

Penjabaan tersebut dapat dibuktikan dengan menjabarkan ekpresi di atas yang akan menghasilkan nilai c_1 sesuai dengan definisi sebelumnya.

Secara garis besar Algoritma Karatsuba memiliki langkah-langkah sebagi berikut :

1. Jika nilai N cukup kecil, hitung fg dengan algoritma naif.
2. Pecah polinomial f menjadi f_0 dan f_1.
3. Pecah polinomial g menjadi g_0 dan g_1.
4. Hitung nilai c_0, c_1, c_2 secara rekursif
5. Bentuk polinomial fg dari nilai c_0, c_1, c_2.

Untuk setiap dua buah polinomial berderajat N yang akan dikalikan, perlu adanya proses mereduksi polinomial tersebut menjadi empat buah polinomial yang dapat diselesaikan dalam kompleksitas waktu $T(N) = 2N$. Setelah tercipta empat buah polinomial, diperlukan adanya proses perhitungan nilai c_0, c_1, c_2 yang dapat dilakukan dengan mengalikan polinomial berderajat $N/2$ sebanyak tiga kali dan melakukan penjumlahan polinomial sebanyak empat kali sehingga kompleksitas waktu untuk proses ini adalah $T(N) = 3T\left(\frac{N}{2}\right) + 4N$. Setelah nilai c_0, c_1, c_2 berhasil dihitung, perlu adanya penggabungan nilai yang dapat dilakukan dengan komplesitas waktu $T(N) = 5N$ yang bersal dari penjumlahan tiga buah polinomial yang memiliki kompleksitas $T(N) = 3N$ dan pengalian dua buah polinomial dengan ekpresi dalam bentuk x^k yang memiliki kompeksitas $T(N) = 2N$.

Berdasarkan perhitungan di atas dapat disimpulkan kompleksitas waktu total dari proses mengalikan dua buah polinomial dengan menggunakan Algoritma Karatsuba adalah

$$T(N) = 2N + 3T\left(\frac{N}{2}\right) + 4N + 5N = 3T\left(\frac{N}{2}\right) + 11N$$

Berdasarkan Teorema Master, kompleksitas waktu Algoritma tersebut adalah $T(N) = O\big(N^{\log_2 3}\big)$.

```
1.  def karatsuba_algorithm(f, g):
2.    # menyamakan ukuran array f dan h
3.    while (len(f) < len(g)):
4.      f.append(0)
5.    while (len(g) < len(f)):
6.      g.append(0)
7.    n = len(f)
8.
9.    # basis, jika polinomial cukup kecil
10.   if (n <= 3):
11.     return naive_algorithm(f,g)
12.
13.   # memecah polinomial
14.   (f0,f1) = reduce(f)
15.   (g0,g1) = reduce(g)
16.
17.   # menghitung nilai c0,c1,c2
18.   c0 = karatsuba_algorithm(f0,g0)
19.   c2 = karatsuba_algorithm(f1,g1)
20.
21.   tmp_0 = sum_polynomial(f0,f1)
22.   tmp_1 = sum_polynomial(g0,g1)
23.   tmp_2 = karatsuba_algorithm(tmp_0,
    tmp_1)
24.   c1 = sum_polynomial(tmp_2,
    neg_polynomial(sum_polynomial(c0,c2)))
```

```
25.
26.    # menggabungkan c0,c1,c2
27.    c1 = [0] * ((n+1)//2) + c1
28.    c2 = [0] * (2*((n+1)//2)) + c2
29.    return sum_polynomial(c0,
     sum_polynomial(c1,c2))
```

Listing 3. Implementasi Algoritma Karatsuba Dalam Bahasa Python

Algoritma ini sedikit lebih baik dibandingkan dengan algoritma naif yang memiliki kompleksitas waktu $O(N^2)$. Meskipun begitu untuk nilai N yang lebih dari 1000 algoritma ini masiih berjalan lambat.

Meskipun algoritma ini memiliki kompleksitas yang lebih baik dibandingkan dengan algoritma naif, akan tetapi untuk nilai N yang kecil, algoritma ini cenderung berjalan lebih lambat dibandingkan dengan algoritma naif. Hal ini disebabkan karena banyaknya overhead pemanggilan fungsi rekursif dan banyaknya proses mereduksi polinomial. Algoritma ini lebih cocok digunakan untuk nilai N yang cukup besar seperti lima ratus. Untuk nilai N dibawah seratus sebaiknya algoritma naif yang digunakan.

C. Fast Fourier Transform

Fast Fourier Transform atau biasa disingkat FFT merupakan suatu algoritma untuk menentukan nilai dari DFT dengan kompleksitas waktu $O(NlogN)$. FFT bekerja dengan konsep *divide and conquer* dengan membagi polinomial menjadi dua buah polinomial yang memiliki derajat setengah dari derajat awalnya. Dengan menggunakan cara naif, untuk menghitung nilai DFT dari suatu polinomial dibutuhkan kompleksitas waktu $O(N^2)$ yang dapat dilakukan dengan cara menghitung setiap suku pada polinomial. Melakukan perhitungan untuk setiap suku pada polinomial membutuhkan kompleksitas waktu $O(N)$, sedangkan untuk menghitung nilai di N titik berbeda dibutuhkan kompleksitas $O(N^2)$. Dengan menggunakan algoritma FFT, proses ini dapat disederhanakan menjadi berkompleksitas $O(NlogN)$.

FFT dapat bekerja karena adanya karakteristik khusus dari bilangan kompleks ω_n sebagai berikut :

$$\omega_n^n = 1$$
$$\omega_n^{n+k} = \omega_n^k$$
$$\omega_n^{\frac{n}{2}} = -1$$
$$\omega_n^{n/2+k} = -\omega_n^k$$

Dengan menggunakan sifat tersebut, proses penggabungan solusi dalam FFT dapat dilakukan menjadi dua kali lebih cepat. Hal tersebut dikarenakan dengan menghitung nilai ω_n^k, nilai ω_n^{n+k} juga dapat terhitung dengan mudah sehingga pada setiap iterasi nilai yang perlu dihitung hanyalah $N/2$.

Untuk sebuah polinomial $f(x)$ dengan jumlah suku merupakan bilangan pangkat dua atau dengan kata lain derajatnya berbentuk $N = 2^k - 1$, dapat dipecah menjadi penjumlahan dua buah polinomial dengan derajat $(N+1)/2$. Untuk polinomial yang berderajat tidak berbentuk pangkat dari dua, polinomial tersebut selalu dapat diperlebar dengan mengisi koefisiennya dengan nol sehingga derajatnya berbentuk dua pangkat.

Pemecahan polinomial tersebut dapat dilakukan sebagai berikut :

$$\begin{aligned} f(x) &= a_0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n \\ &= (a_0 + a_2x^2 + \cdots + a_{n-1}x^{n-1}) \\ &\quad + (a_1x^1 + a_3x^3 + \cdots + a_nx^n) \\ &= g(x^2) + xh(x^2) \end{aligned}$$

dengan

$$g(x) = a_0 + a_2x + \cdots + a_{n-1}x^{\frac{n-1}{2}}$$
$$h(x) = a_1 + a_3x + \cdots + a_nx^{\frac{n-1}{2}-1}$$

Proses menghitung nilai DFT dari f dapat dilakukan dengan cara menghitung nilai DFT dari g dan h kemudian digabungkan dengan persamaan di atas. Untuk menentukan nilai dari suatu polinomial $f(x)$ pada titik x^2, dapat dilakukan dengan mengalikan nilai eksponen dari setiap suku pada polinomial dengan dua. Dalam representasi *array*, untuk menghitung nilai $f(x^2)$ dapat dilakuakan dengan menggeser setiap elemen pada posisi i menjadi posisi $2i$. Hal tersebut dapat dilakukan dengan kompleksitas waktu $O(N)$. Oleh karena itu kompleksitas waktu untuk memecah polinomial adalah $O(N)$.

```
1. def reduce_polynomial(f):
2.    g = []
3.    h = []
4.    for i in range(len(f)):
5.      if (i % 2 == 0):
6.        g.append(f[i])
7.      else:
8.        h.append(f[i])
9.    return (g,h)
```

Listing 4. Implementasi Pemecahan Polinomial Pada Algoritma FFT Dalam Bahasa Python

Secara garis besar, algoritma FFT untuk menghitung nilai DFT dari f adalah sebagai berikut :

1. Jika $N = 1$, kembalikan f
2. Pecah f menjadi g dan h
3. Tentukan nilai DFT dari g dan h
4. Gabungkan nilai DFT dari g dan h untuk mendapatkan nilai DFT dari f

```
1. def fft(f):
2.    if (len(f) <= 1):
3.      return f
4.    n = (2**ceil(log(len(f),2)));
5.    f += [0] * (n-len(f))
6.
7.    (g,h) = reduce_polynomial(f)
8.    _g = fft(g)
9.    _h = fft(h)
10.
11.   result = [0] * n
12.   w = complex(1)
13.   wn = complex(cos(2*PI/n), sin(2*PI/n));
14.   for i in range((n+1)//2):
15.     result[i] = _g[i] + w * _h[i]
16.     result[i+(n+1)//2] =  g[i] - w *  h[i]
```

```
17.     w *= wn
18.
19.   return result
```

Listing 5. Implementasi FFT Dalam Bahasa Python

Langkah nomor dua dapat diselesaikan dalam kompleksitas waktu $O(N)$. Langkah nomor tiga dapat diselesaikan dalam kompleksitas waktu $T(N) = 2T\left(\frac{N}{2}\right)$ dan langkah keempat dapat diselesaikan dalam kompleksitas waktu $O(N)$. Total kompleksitas dari algorita tersebut adalah $T(N) = 2T\left(\frac{N}{2}\right) + O(N)$. Dengan menggunakan Teorema Master, algoritma tersebut dapat disimpulkan memiliki kompleksitas $O(NlogN)$.

D. Inverse DFT

Algoritma FFT dapat menghitung nilai DFT dari suatu polinomial dalam kompleksitas waktu $O(NlogN)$. Seperti yang sudah dipaparkan sebelumnya, dengan mengetahui nilai DFT dari suatu polinomial, bentuk dari polinomial tersebut juga dapat diketahui. Untuk menentukan bentuk polinomial asal dengan menggunakan informasi DFT yang ada, algoritma FFT dapat digunakan. Yang membedakan antara algoritma FFT untuk menghitung nilai DFT dengan algoritma FFT untuk menghitung nilai inverse DFT adalah pada bagian menggabungkan solusi. Pada penggabungan solusi, dasar matriks yang digunakan adalah matriks V^{-1}. Sehingga nilai yang dihitung adalah nilai ω_n^{-k}.

Secara rinci, algoritma inverse FFT adalah sebagai berikut :

```
1. def calc_inverse_withour_scale(f):
2.   if (len(f) <= 1):
3.     return f
4.   n = (2**ceil(log(len(f),2)));
5.   f += [0] * (n-len(f))
6.
7.   (g,h) = reduce_polynomial(f)
8.   _g = calc_inverse_withour_scale(g)
9.   _h = calc_inverse_withour_scale(h)
10.
11.   result = [0] * n
12.   w = complex(1)
13.   wn = complex(cos(2*PI/n), sin(2*PI/n));
14.   for i in range((n+1)//2):
15.     result[i] = (_g[i] + _h[i] / w)
16.     result[i+(n+1)//2] = (_g[i] - _h[i] /
    w)
17.     w *= wn
18.
19.   return result
20.
21.def inverse_fft(f):
22.   f = calc_inverse_withour_scale(f)
23.   for i in range(len(f)):
24.     f[i] /= len(f)
25.   return f
```

Listing 6. Algoritma Inverse FFT Dalam Bahasa Python

E. Perkalian Polinomial Dengan FFT

Secara garis besar, proses perkalian dua buah polinomial f dan g yang berderajat N adalah dengan menghitung nilai $f(x)g(x)$ di N buah titik menggunakan algoritma FFT. Hal ini sama saja dengan menghitung nilai DFT dari $f(x)g(x)$. Dengan mengetahui nilai DFD dari $f(x)g(x)$, bentuk dari polinomial $f(x)g(x)$ juga akan diketahui.

Algoritma FFT untuk mengalikan polinomial adalah sebagai berikut :

```
1. def fft_algorithm(f,g):
2.   if (len(f) < len(g)):
3.     f += [0] * (len(g) - len(f))
4.   elif (len(g) < len(f)):
5.     g += [0] * (len(f) - len(g))
6.
7.   f += [0] * len(f)
8.   g += [0] * len(f)
9.
10.   _f = fft(f)
11.   _g = fft(g)
12.
13.   result = []
14.   for i in range(len(_f)):
15.     result.append(_f[i] * _g[i])
16.   result = inverse_fft(result)
17.   for i in range(len(result)):
18.     result[i] = round(result[i].real)
19.   return result
```

Listing 7. Implementasi Algoritma FFT Pada Perkalian Polinomial

Algoritma FFT sangat cepat dalam waktu eksekusinya. Dilihat dari kompleksitasnya yang $O(NlogN)$ algoritma ini mampu menyelesaikan permasalahan perkalian polynomial dengan derajat mencapai sepuluh ribu dalam waktu kurang lebih satu detik. Untuk nilai derajat yang kecil, algoritma ini masih mampu menyelesaikan perkalian dengan cepat. Meskipun begitu, untuk nilai derajat yang kecil algoritmanaif lebih cocok digunakan karena kebutuhan memorinya yang lebih sedikit. Algoritma FFT umumnya memiliki kebutuhan memori sama seperti dengan Algoritma Karatsuba karena sifatnya yang rekursif.

IV. KESIMPULAN

Berdasarkan hasil eksperimen mengalikan dua buah polinomial dengan tiga macam algoritma, hasil yang diperoleh memberikan bukti bahwa ketiga algoritma yang diuji memiliki tingkat keefektifan yang sama akan tetapi tingkat efisiensinya berbeda. Perbandingan waktu eksekusi ketiga algoritma adalah sebagai berikut :

Algoritma	N				
	3	10	100	1000	10000
Naif	0.0	0.0	0.0025	0.668	32.952
Karatsuba	0.0	0.00	0.002	0.26	23.727
FFT	0.0005	0.001	0.013	0.12	1.79

Tabel 1. Perbandingan Waktu Eksekusi Perkalian Polinomial Dalam Detik

Dari tabel tersebut, dapat dilihat bahwa untuk nilai polinomial yang kecil (derajatnya dibawah seratus), algoritma naif memberikan waktu eksekusi yang lebih cepat dibandingkan dengan algoritma yang menggunakan prinsip *divide and conquer*. Keefisienan algoritma yang berbasis *divide and*

conquer meningkat seiring bertambahnya ukuran N. Hal ini dapat dilihat pada perbedaan waktu eksekusi yang semakin jauh anatara algoritma naif dengan algoritma yang menggunakan prinsip *divide and conquer.*

Jadi, untuk permasalahan perkalian polinomial, algoritma naif dapat digunakan jika derajat dari polinomial yang akan dikalikan relatif kecil yaitu kurang dari seratus. Untuk polinomial dengan derajat kisaran seratus Algoritma Karatsuba dapat digunakan. Sedangkan untuk derajat yang sudah melebihi seribu, Algoritma FFT dapat digunakan.

UCAPAN TERIMA KASIH

Penulis ingin mengucapkan puji syukur kepada Tuhan Yang Maha Esa karena dengan rahmat dan karunia-Nya penulis dapat menyelesaikan makalah berjudul "Penerapan Divide And Conquer Dalam Mengalikan Polinomial" dengan baik. Penulis juga berterima kasih kepada para dosen pengajar mata kuliah IF2211 Strategi Algoritma, Dr. Ir. Rinaldi Munir, M.T.,Masayu Leylia Khodra, S.T., M.T., dan Dr. Nur Ulfa Maulidevi, S.T., M. Sc yang telah memberikan ilmu dan membimbing penulis dalam menjalani perkuliahan sehingga dapat menyelesaikan makalah ini. Penulis juga berterima kasih kepada rekan-rekan yang telah memberikan semangat dan dorongan kepada penulis

REFERENSI

[1] R. Munir, Algoritma DFS dan BFS, 2016. [Online] Tersedia dalam http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2014-2015/BFS%20dan%20DFS%20(2015).pptx. [diakses 19 Mei 2017 pukul 06.13 WIB].

[2] Yan-Bin Jia. Polynomial Multiplication and Fast Fourier Transform. 18 Mei 2017. http://web.cs.iastate.edu/~cs577/handouts/polymultiply.pdf

[3] K.T. Leung, I.A.C. Mok, S.N. Suen. "Polynomials And Equations,". Hong Kong University Press. 1992.

[4] Polynomial Function. 18 Mei 2017. http://www.augusta.k12.va.us/cms/lib01/va01000173/centricity/domain/766/chap07.pdf

[5] https://people.eecs.berkeley.edu/~vazirani/algorithms/chap2.pdf

[6] Divide-and-Conquer Algorithm. 18 Mei 2017. https://people.eecs.berkeley.edu/~vazirani/algorithms/chap2.pdf

PERNYATAAN

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, 19 Mei 2017

Jauhar Arifin / 13515049