Analisis Perbandingan Efisiensi Metode Divide and Conquer dan Dynamic Programming pada Penyelesaian Persoalan Relasi Rekurens Linear

Farhan Nabil Suryono - 13521114
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13521114@std.stei.itb.ac.id

***Abstract*—Persoalan Relasi Rekurens Linear adalah salah satu persoalan yang sering muncul dalam analisis algoritma dan matematika diskrit. Persoalan ini dapat dikerjakan dengan beberapa pendekatan, beberapa diantaranya adalah metode *Dynamic Programming* dan *Divide and Conquer*. Dari hasil implementasi, pendekatan *Dynamic Programming* memiliki kompleksitas waktu $O(nk)$ dan kompleksitas ruang $O(n)$, sedangkan pendekatan *Divide and Conquer* memiliki kompleksitas waktu $O(k^3 log(n))$ dan kompleksitas ruang $O(k^2 log(n))$ dengan k adalah tingkat relasi rekurens.**

Keywords—Linear Recurrence Relation, Divide and Conquer, Dynamic Programming

I. Pendahuluan

Persoalan relasi rekurens merupakan salah satu persoalan yang sering muncul dalam analisis algoritma dan matematika diskrit. Relasi rekurens berbentuk suatu barisan yang memiliki sifat dimana nilai suku pada barisan tersebut bergantung pada nilai dari suku-suku sebelumnya.

Relasi rekurens linear adalah *subset* dari relasi rekurens dimana setiap sukunya hanya bergantung secara linear terhadap suku-suku sebelumnya. Salah satu contoh yang terkenal dari relasi rekurens linear adalah barisan Fibonacci dimana nilai suatu suku berupa penjumlahan nilai dari kedua suku sebelumnya.

Ada beberapa pendekatan yang dapat digunakan untuk menyelesaikan persoalan relasi rekurens linear seperti *brute force*, *divide and conquer*, dan *dynamic programming*.

Pendekatan *brute force* pada umumnya merupakan pendekatan yang pertama kali terpikirkan dalam melakukan penyelesaian persoalan. Namun, pada persoalan relasi rekurens, pendekatan *brute force* sangatlah lambat karena jumlah perhitungan dapat tumbuh secara eksponensial seiring dengan membesarnya suku yang ingin dicari. Akibatnya, program akan menjadi tidak efektif ketika ingin mencari nilai dari suku-suku yang besar.

Oleh karena itu, dalam makalah ini, penulis akan membahas alternatif lain penyelesaian relasi rekurens linear, yaitu dengan *divide and conquer* dan *dynamic programming*. Penulis juga akan menganalisis efisiensi program dari segi waktu eksekusi serta penggunaan memori. Penulis juga akan membahas kelebihan serta kekurangan kedua pendekatan.

II. Landasan Teori

A. Relasi Rekurens Linear

Relasi rekurens linear adalah salah satu jenis barisan (*sequence*) yang merupakan bagian dari relasi rekurens. Relasi rekurens sendiri memiliki sifat utama yaitu setiap sukunya memiliki nilai yang bergantung pada nilai elemen-elemen sebelumnya. Relasi rekurens linear adalah relasi rekurens yang nilai sukunya bergantung hanya secara linear terhadap nilai elemen sebelumnya.

Dalam relasi rekurens linear, dikenal beberapa istilah seperti tingkat (*order*) dan basis. Tingkat dari suatu relasi rekurens menunjukkan jumlah suku sebelumnya yang digunakan untuk menghitung nilai suatu suku. Basis adalah nilai fungsi yang telah terdefinisi.

Sebagai contoh, persamaan relasi rekurens linear yang paling terkenal adalah bilangan Fibonacci yang didefinisikan dengan persamaan berikut.

$$f_n = f_{n-1} + f_{n-2}$$
$$f_0 = 0$$
$$f_1 = 1$$

Persamaan diatas menunjukkan bahwa suku ke-n pada barisan merupakan penjumlahan 2 suku sebelumnya yang berarti bilangan Fibonacci adalah relasi rekurens linear dengan *order* 2. Dalam bilangan Fibonacci, terdapat 2 basis yaitu suku ke-0 bernilai 0 dan suku ke-1 bernilai 1.

B. Divide and Conquer

Divide and Conquer adalah salah satu pendekatan yang dapat digunakan untuk menyelesaikan suatu persoalan secara efektif. Pendekatan ini terdiri atas 3 langkah utama yang bekerja bersama untuk menyelesaikan persoalan.

Langkah pertama adalah *divide* yaitu membagi persoalan menjadi beberapa sub-persoalan yang memiliki kemiripan

dengan persoalan awal namun memiliki ukuran yang lebih kecil. Langkah kedua adalah *Conquer* yaitu setiap sub-persoalan yang telah dibagi diselesaikan. Langkah terakhir adalah *Combine* dimana solusi dari seluruh sub-persoalan digabung sehingga membentuk solusi dari persoalan awal.

C. *Dynamic Programming*

Program Dinamis atau *Dynamic Programming* adalah suatu pendekatan pemecahan persoalan dengan menguraikan solusi menjadi sekumpulan tahapan yang memiliki *state* tertentu. Solusi persoalan dari metode *dynamic programming* akan menghasilkan serangkaian keputusan yang saling berkaitan.

Dynamic Programming umumnya digunakan untuk persoalan optimasi (maksimasi atau minimisasi). *Dynamic Programming* menggunakan prinsip optimalitas yang menyatakan bahwa apabila suatu solusi total optimal, maka bagian solusi dari tahap ke-k juga optimal.

Ada dua pendekatan yang digunakan pada *Dynamic Programming* yaitu:

1. Top-down

   Perhitungan dilakukan dari tahap 1, 2, …, n – 1, n. Umumnya dikerjakan secara iteratif.

2. Bottom-up

   Perhitungan dilakukaan dari tahap n, n – 1, …, 2, 1. Umumnya dikerjakan secara rekursif.

III. PEMBAHASAN

A. *Representasi Relasi Rekurens Linear dalam Program*

Relasi rekurens linear yang akan diselesaikan memiliki bentuk sebagai berikut.

$$f_n = c_1 f_{n-1} + c_2 f_{n-2} + \cdots + c_k f_{n-k} + C$$

Dengan k adalah *order* dari relasi tersebut dan C adalah suatu konstanta. Dalam program, relasi rekurens linear dibuat dengan langkah-langkah sebagai berikut.

1. Program meminta nilai k sebagai *order* relasi.
2. Program meminta 1 angka yang akan menjadi konstanta relasi.
3. Program meminta k angka yang merepresentasikan koefisien (c_1, c_2, …, c_k) yang akan disimpan pada suatu *array* bernama *coefficients* bersama dengan nilai konstanta dengan indeks ke-0 adalah konstanta dan indeks ke-i adalah c_i.
4. Program meminta k angka yang merepresentasikan basis relasi rekurens yaitu (f(0), f(1), …, f(k – 1)) dan menyimpannya dalam suatu *array* bernama *baseValues*.

B. *Penyelesaian dengan Metode Dynamic Programming*

Penyelesaian relasi rekurens linear dengan metode *Dynamic Programming* dapat dilakukan dengan langkah-langkah sebagai berikut.

1. Apabila nilai n lebih kecil dari k, nilai *coefficients*[n] akan langsung dikembalikan. Apabila n ≥ k maka lanjut ke langkah 2.
2. Inisialisasi *array* kosong bernama *dp* dengan panjang n + 1. Isi dari *array* dengan indeks ke-i akan digunakan untuk merepresentasikan nilai dari f_i.
3. Isi *dp* mulai dari indeks ke-0 hingga indeks ke-(k – 1) dengan basis relasi rekurens.
4. Program melakukan iterasi indeks *i* dengan *range* [k..n]. Dalam iterasi ke-i, program akan menghitung nilai f_i dan menyimpannya pada *dp*[i].
5. Program mengembalikan nilai *dp*[n] yang merupakan nilai dari f_n.

Berikut adalah hasil implementasi dalam bahasa Python.

```
def solve(n: int, coefficient: Coefficients, baseValue: BaseValues)
-> Decimal:
    order: int = len(baseValue)

    if n < order:
        return baseValue[n]

    dp: List[Decimal] = []
    for i in range(n + 1):
        if i < order:
            dp.append(baseValue[i])
        else:
            value: Decimal = Decimal(0)

            for j in range(1, len(coefficient)):
                value += dp[i - j] * coefficient[j]
                value %= int(1e9)

            value += coefficient[0]
            value %= int(1e9)

            dp.append(value)

    return dp[n]
```

Berdasarkan implementasi di atas, penyelesaian menggunakan 2 *nested loop* yang masing-masing memiliki

jumlah perulangan sebanyak nilai n dan tingkat relasi. Maka dari itu, kompleksitas waktu dari algoritma ini adalah $O(nk)$.

Selain itu, implementasi menggunakan *array of Decimal* yang berukuran n sehingga kompleksitas ruang dari algoritma ini adalah $O(n)$.

C. Penyelesaian dengan Metode Divide and Conquer

Penyelesaian relasi rekurens linear dengan *Divide and Conquer* sedikit lebih rumit dibandingkan dengan *Dynamic Programming*. Penyelesaian ini akan menggunakan suatu teknik khusus yang sering disebut dengan *Matrix Exponentiation*.

Konsep ini dimulai dari pengamatan berikut. Apakah ada suatu matriks M yang akan selalu memenuhi persamaan berikut

$$\begin{bmatrix} f_n \\ f_{n-1} \\ f_{n-2} \\ ... \\ f_{n-k} \end{bmatrix} = M \times \begin{bmatrix} f_{n-1} \\ f_{n-2} \\ f_{n-3} \\ ... \\ f_{n-k-1} \end{bmatrix}$$

Dengan observasi, didapat bahwa matriks M berikut akan selalu memenuhi persamaan di atas.

$$M = \begin{bmatrix} c_1 & c_2 & ... & c_{k-1} & c_k \\ 1 & 0 & ... & 0 & 0 \\ 0 & 1 & ... & 0 & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & ... & 1 & 0 \end{bmatrix}$$

Namun, matriks di atas baru dapat digunakan untuk menyelesaikan relasi rekurens homogen linear (tidak memiliki konstanta). Dengan sedikit modifikasi maka didapat hubungan berikut

$$\begin{bmatrix} f_n \\ f_{n-1} \\ f_{n-2} \\ ... \\ f_{n-k} \\ C \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & ... & c_{k-1} & c_k & 1 \\ 1 & 0 & ... & 0 & 0 & 0 \\ 0 & 1 & ... & 0 & 0 & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & 1 & 0 & 0 \\ 0 & 0 & ... & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} f_{n-1} \\ f_{n-2} \\ f_{n-3} \\ ... \\ f_{n-k-1} \\ C \end{bmatrix}$$

Dengan ini, kita mendapatkan suatu matriks yang apabila dikalikan dengan matriks berisi suku n – k – 1 hingga n – 1 dari suatu relasi rekurens linear maka kita akan mendapatkan suku berikutnya di setiap baris yaitu n – k hingga n.

Dapat dilihat bahwa matriks M akan selalu bernilai tetap untuk suatu relasi rekurens linear maka persoalan relasi rekurens linear dapat diselesaikan dengan cara berikut.

$$\begin{bmatrix} f_n \\ f_{n-1} \\ f_{n-2} \\ ... \\ f_{n-k} \\ C \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & ... & c_{k-1} & c_k & 1 \\ 1 & 0 & ... & 0 & 0 & 0 \\ 0 & 1 & ... & 0 & 0 & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & 1 & 0 & 0 \\ 0 & 0 & ... & 0 & 0 & 1 \end{bmatrix}^{n-k} \times \begin{bmatrix} f_{k-1} \\ f_{k-2} \\ f_{k-3} \\ ... \\ f_0 \\ C \end{bmatrix}$$

Sebagai contoh, bilangan fibonacci adalah suatu relasi rekurens linear dengan *order* 2 dengan persamaan berikut.

$$f_n = f_{n-1} + f_{n-2}$$
$$f_0 = 0$$
$$f_1 = 1$$

Maka, bilangan fibonacci ke-n bisa diselesaikan dengan persamaan matriks di bawah.

$$\begin{bmatrix} f_n \\ f_{n-1} \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{n-2} \times \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Berbekal teknik yang telah dijelaskan di atas maka kita bisa menggunakan *Divide and Conquer* pada proses perpangkatan matriksnya. Mendapatkan matriks berpangkat n dengan *Divide and Conquer* memiliki cara yang sama dengan bilangan biasa yaitu dengan proses rekursif sebagai berikut.

1. Apabila n bernilai 1, kembalikan matriks awal
2. Apabila n bernilai genap, kembalikan hasil perkalian matriks pangkat n div 2 dengan dirinya sendiri.
3. Apabila n bernilai ganjil, kembalikan hasil perkalian matriks pangkat n div 2 dengan dirinya sendiri dan matriks awal.

Berikut adalah hasil implementasi dalam bahasa Python.

```
def matrixMultiply(mat1: DecimalMatrix, mat2: DecimalMatrix) -
> DecimalMatrix:
    row1, col1 = len(mat1), len(mat1[0])
    row2, col2 = len(mat2), len(mat2[0])

    result = [[Decimal(0) for _ in range(col2)] for _ in range(row1)]
    for i in range(row1):
        for j in range(col2):
            for k in range(col1):
                result[i][j] += mat1[i][k] * mat2[k][j]
                result[i][j] %= int(1e9)

    return result
```

```
def getExponentMatrix(matrix: DecimalMatrix, power: int) ->
DecimalMatrix:
    if power == 1:
        return matrix
    else:
        halfPowerMatrix: DecimalMatrix =
getExponentMatrix(matrix, power // 2)
        if power % 2 == 0:
            return matrixMultiply(halfPowerMatrix, halfPowerMatrix)
        else:
            return matrixMultiply(matrixMultiply(halfPowerMatrix,
halfPowerMatrix), matrix)


def solve(n: int, coefficient: Coefficients, baseValue: BaseValues)
-> Decimal:
    order: int = len(baseValue)

    if n < order:
        return baseValue[n]

    baseMatrix: DecimalMatrix = []
    for i in range(order - 1, -1, -1):
        baseMatrix.append([baseValue[i]])
    baseMatrix.append([coefficient[0]])

    identityMatrix: DecimalMatrix = []
    for i in range(order + 1):
        if i == 0:
            identityMatrix.append(coefficient[1:] + [Decimal(1)])
        elif i == order:
            identityMatrix.append([Decimal(0)] * order +
[Decimal(1)])
        else:
            identityMatrix.append([Decimal(0)] * (order + 1))
            identityMatrix[i][i - 1] = Decimal(1)

    resultMatrix: DecimalMatrix =
matrixMultiply(getExponentMatrix(identityMatrix, n - order + 1),
baseMatrix)
    return resultMatrix[0][0] % int(1e9)
```

Berdasarkan implementasi di atas, penyelesaian menggunakan kalkulasi terberat berada pada perhitungan matriks eksponen. Perkalian matriks disini menggunakan *brute force* yang memiliki kompleksitas waktu $O(k^3)$ karena matriks berukuran sebanyak k x k. Perkalian matriks ini dilakukan sebanyak fungsi getExponentMatrix dipanggil yang memiliki kompleksitas $O(\log n)$. Maka dari itu, kompleksitas waktu dari algoritma ini adalah $O(k^3 \log n)$.

Selain itu, implementasi menggunakan matriks berukuran k x k yang dibuat setiap fungsi eksponen dipanggil sehingga kompleksitas ruang dari algoritma ini adalah $O(k^2 \log n)$.

IV. Analisis

Pada uji coba yang penulis lakukan, nilai koefisien akan dibatasi antara 1 hingga 20 yang di-*generate* secara acak agar tidak terlalu menambah beban pada program dan hanya berfokus pada algoritma.

Kasus uji yang digunakan untuk analisis di makalah ini dapat dilihat pada *github* pada lampiran di dalam folder test.

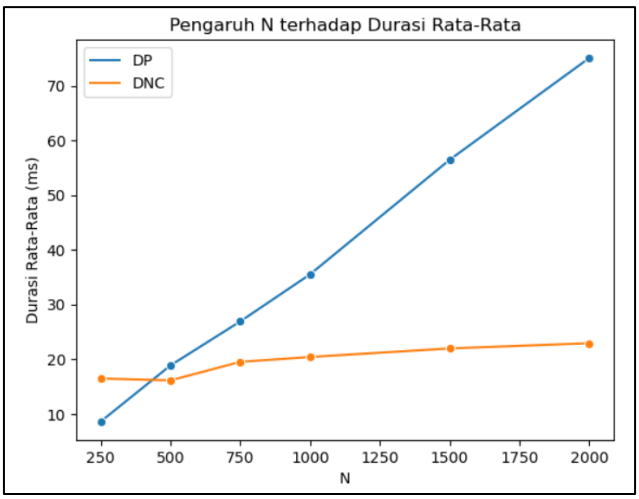
A. *Analisis Pengaruh Nilai n*

Variabel nilai n melambangkan suku ke berapa pada suatu relasi rekurens linear. Semakin besar nilai n maka perhitungan yang perlu dilakukan oleh komputer akan semakin banyak.

Uji coba pengaruh nilai n ini akan menggunakan spesifikasi sebagai berikut.

- 50 *test case* dengan tingkat relasi yang tetap yaitu 5.
- Untuk setiap *test case*, program akan menghitung f(n) dengan n bernilai 250, 500, 750, 1000, 1500, dan 2000 menggunakan kedua algoritma
- Hasil *testing* akan disimpan dengan menyimpan nilai n, durasi dalam milisekon, dan memori yang digunakan dalam bytes.

Berikut adalah grafik visualisasi hasil *plotting* durasi waktu yang dibutuhkan program untuk menyelesaikan perhitungan dengan kedua algoritma terhadap nilai n.

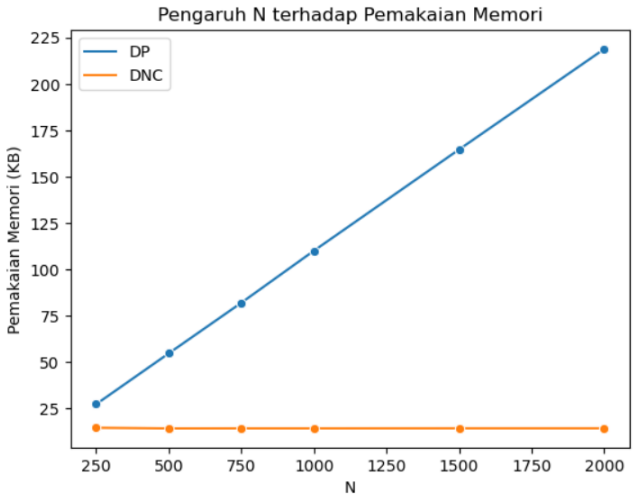


Gambar 1 Grafik Durasi Rata-Rata terhadap Nilai N

Sumber: Dokumen Pribadi

Dari grafik di atas, terlihat bahwa pertumbuhan waktu yang dibutuhkan algoritma *Dynamic Programming* lebih cepat membesar dibanding algoritma *Divide and Conquer*. Hal ini sesuai dengan kompleksitas waktu masing-masing dimana kompleksitas waktu *Dynamic Programming* berbanding lurus dengan nilai n dan kompleksitas waktu *Divide and Conquer* membesar berbanding lurus dengan nilai log(n).

Lalu, berikut adalah grafik visualisasi hasil *plotting memory* yang digunakan program untuk menyelesaikan perhitungan dengan kedua algoritma terhadap nilai n.



Gambar 2 Grafik Pemakaian Memori terhadap Nilai N

Sumber: Dokumen Pribadi

Dari grafik di atas, terlihat bahwa algoritma *Dynamic Programming* jauh lebih boros memori dibandingkan *Divide and Conquer*. Hal ini sesuai dengan kompleksitas ruang masing-masing dimana kompleksitas ruang *Dynamic Programming* berbanding lurus dengan nilai n dan kompleksitas ruang *Divide and Conquer* hanya berbanding lurus dengan log(n).

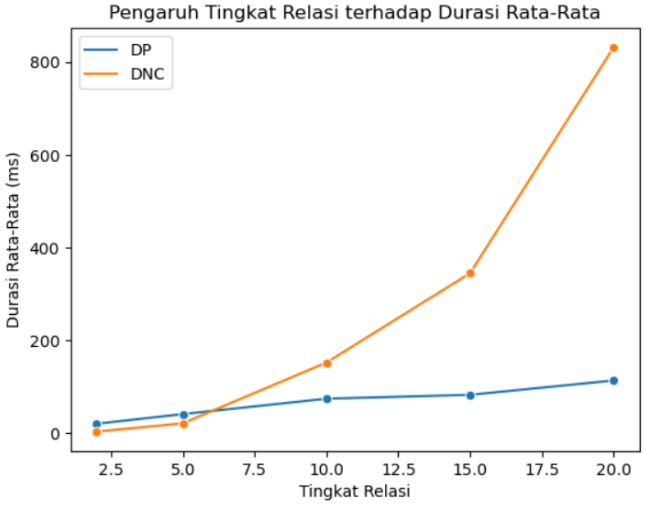
B. *Analisis Pengaruh Tingkat Relasi*

Variabel tingkat relasi (k) menyatakan suku sebelum ke berapa yang diperlukan untuk menghitung nilai suatu suku di relasi rekurens linear.

Uji coba pengaruh tingkat relasi ini akan menggunakan spesifikasi sebagai berikut.

- 50 *test case* dimana tingkat relasinya bervariasi di antara 2, 5, 10, 15, dan 20. Masing-masing tingkat relasi memiliki 10 *test case*.
- Untuk setiap *test case*, program akan menghitung nilai dari f(1000) menggunakan kedua algoritma.
- Hasil *testing* akan disimpan dengan menyimpan tingkat relasi, durasi dalam milisekon, dan memori yang digunakan dalam bytes.

Berikut adalah grafik visualisasi hasil *plotting* durasi waktu yang dibutuhkan program untuk menyelesaikan perhitungan dengan kedua algoritma terhadap tingkat relasi.

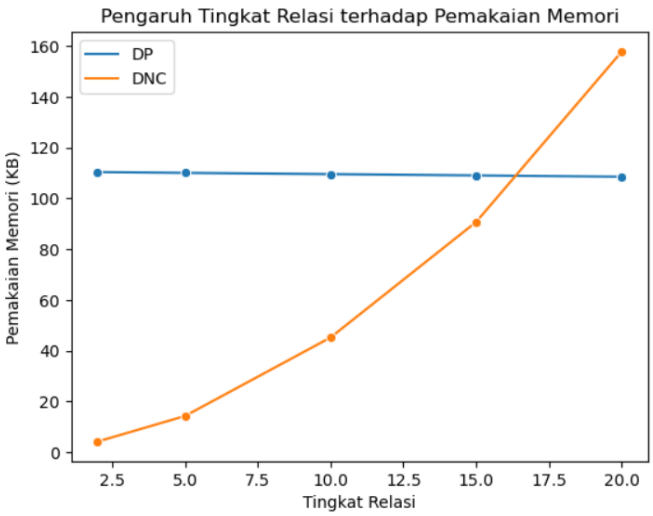


Gambar 3 Grafik Durasi Rata-Rata terhadap Tingkat Relasi

Sumber: Dokumen Pribadi

Dari grafik di atas, terlihat bahwa pertumbuhan waktu yang dibutuhkan algoritma *Divide and Conquer* lebih cepat membesar dibanding algoritma *Dynamic Programming*. Hal ini sesuai dengan kompleksitas waktu masing-masing dimana kompleksitas waktu algoritma *Dynamic Programming* berbanding lurus dengan nilai k dan kompleksitas waktu *Divide and Conquer* berbanding lurus dengan nilai k^3.

Lalu, berikut adalah grafik visualisasi hasil *plotting memory* yang digunakan program untuk menyelesaikan perhitungan dengan kedua algoritma terhadap tingkat relasi.



Gambar 2 Grafik Pemakaian Memori terhadap Tingkat Relasi

Sumber: Dokumen Pribadi

Dari grafik di atas, terlihat bahwa memori yang digunakan *Divide and Conquer* di awal kecil namun cepat membesar, sedangkan memori yang digunakan *Dynamic Programming* selalu konstan. Hal ini sesuai dengan kompleksitas ruang masing-masing dimana kompleksitas ruang *Dynamic Programming* sama sekali tidak dipengaruhi tingkat relasi, sedangkan kompleksitas ruang *Divide and Conquer* berbanding lurus dengan k^2.

V. KESIMPULAN

Penyelesaian persoalan relasi rekurens linear dapat menggunakan dua pendekatan yaitu *Dynamic Programming* dan *Divide and Conquer*.

Berdasarkan hasil analisis, pendekatan *Divide and Conquer* lebih unggul dalam kecepatan dan pemakaian memori dalam menyelesaikan relasi rekurens dengan n tinggi pada relasi rekurens linear dengan tingkat rendah yang merupakan kasus yang umum terjadi.

Namun, pada kasus dimana tingkat relasi relatif tinggi, pendekatan *Dynamic Programming* lebih baik karena pemakaian memori yang tidak terpengaruh tingkat relasi dan kompleksitas waktu yang jauh lebih rendah terhadap tingkat relasi.

Selain itu, dengan sedikit modifikasi, *Dynamic Programming* dapat mengembalikan *array* solusi sehingga lebih baik apabila ingin dicari untuk banyak nilai n dalam relasi rekurens linear yang sama.

Oleh karena itu, dapat disimpulkan bahwa dalam menyelesaikan relasi rekurens linear yang memiliki tingkat tidak terlalu tinggi, penggunaan pendekatan *Divide and Conquer* jauh lebih baik. Namun, pada relasi rekurens linear dengan tingkat relasi yang tinggi, *Dynamic Programming* bisa jadi kandidat yang lebih diunggulkan karena pertumbuhannya yang tidak terlalu dipengaruhi besarnya tingkat relasi. *Dynamic Programming* juga dapat digunakan untuk mencari beberapa f(n) sekaligus untuk relasi yang sama.

VI. UCAPAN TERIMA KASIH

Penulis disini ingin menyampaikan ucapan terima kasih kepada semua pihak yang telah membantu penulis dalam penyusunan makalah yang berjudul "Analisis Perbandingan Efisiensi Metode Divide and Conquer dan Dynamic Programming pada Penyelesaian Persoalan Relasi Rekurens Linear" sebagai tugas akhir mata kuliah IF2211 Strategi Algoritma, khususnya kepada:

1. Tuhan yang Maha Esa karena atas nikmat, rahmat, dan karunia-Nya, penulis dapat menyelesaikan makalah ini.
2. Keluarga penulis, khususnya kepada kedua orang tua yang senantiasa memberikan dukungan kepada penulis.
3. Dr. Nur Ulfa Maulidevi ST, M.Sc., selaku dosen pengajar mata kuliah IF2211 Strategi Algoritma yang telah mengajar dan memberi banyak ilmu untuk penyelesaian makalah ini.
4. Dr. Ir. Rinaldi Munir, M.T. yang telah menyimpan semua bahan pembelajaran mata kuliah IF2211 Strategi Algoritma di website beliau yang sangat membantu pengerjaaan makalah ini.
5. Teman-teman yang telah banyak membantu penulis menjalani perkuliahan.

VII. REFERENSI

[1] Halim, Steven, & Halim, Felix. (2013). *Competitive Programming 3*.

[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Rekursi-dan-relasi-rekurens-(Bagian-1).pdf diakses pada 20 Mei 2023

[3] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Rekursi-dan-relasi-rekurens-(Bagian-2).pdf diakses pada 20 Mei 2023

[4] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-(2021)-Bagian1.pdf diakses pada 20 Mei 2023

[5] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf diakses pada 20 Mei 2023

[6] https://study.com/learn/lesson/recurrence-relation-examples-formula.html diakses pada 20 Mei 2023

[7] https://www.geeksforgeeks.org/matrix-exponentiation diakses pada 20 Mei 2023

[8] https://zobayer.blogspot.com/2010/11/matrix-exponentiation.html diakses pada 20 Mei 2023

VIII. LAMPIRAN

Repository GitHub: https://github.com/Altair1618/DnC-DP-LinearRecurrence

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, 22 Mei 2023

Farhan Nabil Suryono
13521114