Berbagai Kompleksitas Algoritma Pencari Suku ke-N Barisan Fibonacci dari Rekursif, *Dynamic Programming*, hingga Solusi Rekurens

Widya Anugrah Putra 13519105[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13519105@std.stei.itb.ac.id

***Abstract*—Barisan Fibonacci adalah barisan bilangan 0,1,1,2,3,5,8,13,21,34,... yang mempunyai relasi rekurens antara suku-sukunya, yaitu suatu suku tertentu terbentuk dari hasil penjumlahan 2 suku sebelumnya. Banyak cara untuk menentukan suku ke-N dari barisan fibonacci, mulai dari solusi naif secara rekursif, *dynamic programming*, hingga menggunakan solusi rekurens. Tentu kompleksitas algoritmanya berbeda-beda, sehingga pemrogram diharapkan mampu memilih algoritma yang efisien.**

***Keywords*— Barisan Fibonacci, *Dynamic Programming*, Kompleksitas Algoritma ,Notasi O Besar, Rekurens.**

I. PENDAHULUAN

Suatu permasalahan dapat diselesaikan dengan banyak cara/algoritma. Contohnya pada persoalan menghitung suku ke-N dari barisan fibonacci. Namun, tidak semua algoritma yang bisa dipakai efisien, karena kompleksitas algoritmanya berbeda-beda, sehingga waktu yang dibutuhkan pun berbeda pula.

Kompleksitas algoritma adalah besaran waktu dan ruang yang dibutuhkan untuk menyelesaikan permasalahan. Semakin kompleks algoritma yang digunakan, maka semakin besar pula waktu dan ruang yang dibutuhkan. Pada tulisan ini kita hanya akan sedikit menyinggung kompleksitas ruang dan fokus pada kompleksitas waktunya saja.

Untuk menunjukkan bahwa seorang *programmer* harus bisa mengaplikasikan algoritma yang tepat dan efisien, pada tulisan ini akan dibahas algoritma-algoritma untuk mencari suku ke-N dari barisan fibonacci menggunakan solusi naif, *dynamic programming* dan solusi rekurens.

II. LANDASAN TEORI

A. Rekursi dan Relasi Rekurens

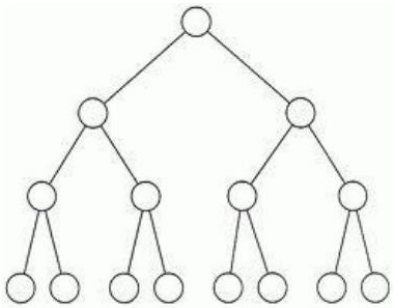
Jika sebuah objek didefinisikan dalam terminologi dirinya sendiri maka ia dikatakan rekursif. Rekursi adalah proses mendefinisikan suatu objek dalam terminologi dirinya sendiri.

Fungsi rekursif adalah fungsi yang terdiri dari langkah basis dan langkah rekurens. Langkah basis berisi nilai fungsi yang terdefinisikan secara eksplisit, dan langkah ini juga menghentikan proses rekursif yang terjadi di langkah rekurens. Langkah rekurens mendefinisikan fungsi dalam terminologi dirinya sendiri, dalam langkah ini proses rekurens terjadi. Contohnya fungsi faktorial secara rekursif didefinisikan sebagai berikut

$$n! = \begin{cases} 1 & , n = 0 \\ n \cdot (n-1)! & , n > 0 \end{cases}$$

Gambar 1 Fungsi faktorial rekursif (Sumber : http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Relasi-dan-Fungsi-Bagian1-(2020).pdf)

Struktur rekursif adalah struktur yang bentuk seluruh strukturnya melakukan perulangan di bagian strukturnya. Contoh struktur rekursif yang berguna di bidang komputer adalah *binary tree*.



Gambar 1 Struktur Binary Tree (Sumber : http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Relasi-dan-Fungsi-Bagian1-(2020).pdf)

Barisan rekursif memiliki relasi rekurens. Relasi rekurens adalah persamaan yang mengekspresikan suku secara rekursif menggunakan satu atau lebih suku sebelumnya. Contoh barisan rekursif adalah barisan fibonacci 0,1,1,2,3,5,8,13,... yang dapat dinyatakan dengan relasi rekurens $f_n = f_{n-1} + f_{n-2}, f_0 = 0$ dan $f_1 = 1$.

Sebuah formula yang tidak melibatkan lagi *term* rekursif dari relasi rekurens disebut sebagai solusi rekurens. Solusi rekurens dapat diperoleh dengan cara iteratif atau dengan metode yang sistematis memanfaatkan relasi rekurens yang berbentuk homogen lanjar.

Penyelesaian dengan metode iteratif dilakukan dengan mengubah rekurens menjadi penjumlahan. Kita melakukan

iterasi hingga mencapai kondisi/suku awal (basis), sehingga formula yang didapatkan mengandung *term* suku basis.

Relasi rekurens dikatakan berbentuk homogen lanjar jika berbentuk

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k}$$

yang dalam hal ini $c_1, c_2, \ldots, c_k$ adalah bilangan riil dan $c_k \neq 0$. Solusi relasi rekurens yang berbentuk homogen lanjar adalah mencari bentuk $a_n = r^n$ yang dalam hal ini r adalah konstanta. Sulihkan kedua persamaan tadi dan didapatkan

$$r^n = c_1 r^{n-1} + c_2 r^{n-2} + \ldots + c_k r^{n-k}$$

Lalu, bagi kedua ruas dengan r^{n-k} didapatkan

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \ldots - c_k = 0$$

Persamaan terakhir disebut sebagai persamaan karakteristik dari relasi rekurens dan solusi-solusinya disebut sebagai akar-akar karakteristik dan merupakan komponen dari solusi relasi rekurens yang sedang kita cari ($a_n = r^n$). Sehingga kita akan mempunyai $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n + \ldots + \alpha_k r_k^n$ untuk n = 0,1,2,... dengan α_n berupa konstanta.

Untuk relasi rekurens dengan derajat k = 2, maka kita punya

$$a_n = c_1 a_{n-1} + c_2 a_{n-2}$$

Sehingga persamaan karakteristiknya adalah

$$r^2 - c_1 r - c_2 = 0$$

dengan akar-akar persamaannya adalah r_1 dan r_2. Jika kedua akarnya berbeda maka berlaku $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$, sedangkan jika kedua akarnya kembar ($r_1 = r_2 = r_0$) maka berlaku $a_n = \alpha_1 r_0^n + \alpha_2 n r_0^n$.

B. Kompleksitas Algoritma

Kompleksitas algoritma adalah besaran yang dipakai untuk menerangkan model abstrak pengukuran waktu dan ruang yang dibutuhkan sebuah algoritma untuk bekerja. Semakin kompleks algoritma yang dipakai, maka semakin besar pula waktu dan ruang yang digunakan. Kompleksitas algoritma dibagi menjadi 2 yaitu kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu, $T(n)$, diukur dari banyaknya komputasi/operasi khas (tipikal) yang mendasari algoritma. Kompleksitas ruang, $S(n)$, diukur dari banyaknya memori yang dipakai agar algoritma dapat berjalan. Namun, dalam penghitungan kompleksitas waktu, kita lebih tertarik pada laju perubahan kebutuhan waktu sebuah algoritma jika masukan (n) meningkat. Oleh karena itu, kita lebih sering menuliskannya dalam notasi O-besar (Big-O Notation).

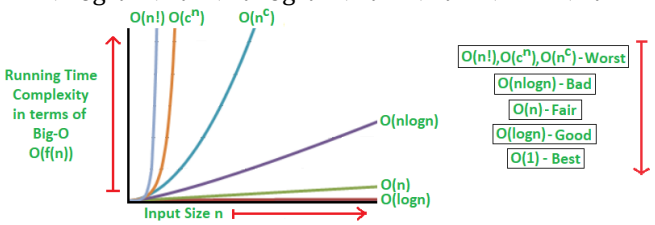
Notasi O-besar ditulis dengan bentuk $O(n)$, dan defnisinya $T(n) = O(f(n))$ bila terdapat konstanta C dan n_0 sedemikian sehingga $T(n) \leq C\, f(n)$ untuk $n \geq n_0$. Sehingga $f(n)$ bisa dianggap sebagai batas lebih atas dari $T(n)$ untuk n yang besar.

Kelompok Algoritma	Nama
$O(1)$	konstan
$O(\log n)$	logaritmik
$O(n)$	linier
$O(n \log n)$	linier logaritmik
$O(n^2)$	kuadratik
$O(n^3)$	kubik
$O(2^n)$	eksponensial
$O(n!)$	faktorial

Tabel 1 Beberapa pengelompokan kompleksitas algoritma

Urutan kompleksitas waktu algoritma pada tabel di atas adalah

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$



Gambar 3 Chart laju pertumbuhan kompleksitas algoritma (Sumber : https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/)

1. $O(1)$.

Kompleksitas $O(1)$ artinya waktu yang dibutuhkan algoritma tetap, tidak bergantung pada besarnya masukan. Algoritma yang memiliki kompleksitas ini biasanya tidak memiliki perulangan. Contoh algoritma pertukaran x dan y sebagai berikut:

```
x = x + y;
y = x - y;
x = x - y;
```

Gambar 4 Pertukaran dua buah integer (Sumber: koleksi pribadi)

2. $O(\log n)$

Kompleksitas $O(\log n)$ artinya laju pertumbuhan waktu yang dibutuhkan sebanding dengan logaritma dari besarnya masukan. Algoritma yang mempunyai kompleksitas ini biasanya algoritma yang memecah masalah besar menjadi masalah kecil lalu menyelesaikan masalah-masalah kecil. Contoh algoritma sederhana yang memiliki kompleksitas ini adalah fungsi power secara rekursif ini.

```
int Power(int x, int n){
    if (x == 0) return 0;
    else if (n == 0) return 1;
    else if (n % 2 == 0) return Power(x*x,n/2);
    else return Power(x*x,n/2) * x;
}
```

Gambar 5 Power (perpangkatan) dengan rekursif (Sumber : koleksi pribadi)

3. $O(n)$

Kompleksitas ini berarti laju pertumbuhan waktunya sebanding dengan besarnya masukan yang diterima. Contoh algoritma sederhana yang memiliki kompleksitas ini adalah fungsi yang menghitung jumlah N bilangan asli pertama.

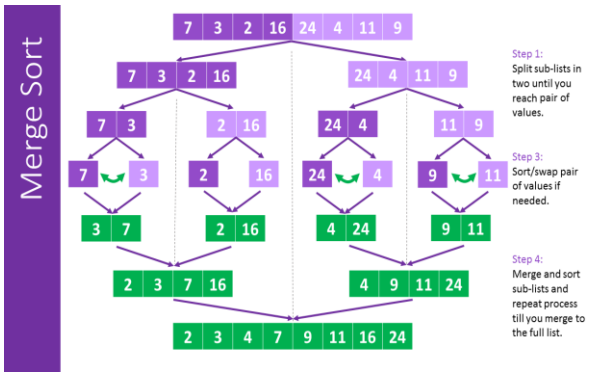
```
int Nsum (int N){
    int hasil = 0;
    for (int i = 1; i <= N; ++i){
        hasil = hasil + i;
    }
    return hasil;
}
```

Gambar 6 Menghitung N bilangan asli pertama (Sumber : koleksi pribadi)

4. $O(n \log n)$

Algoritma yang memiliki kompleksitas ini biasanya memecah permasalah menjadi bagian-bagian yang lebih kecil,

lalu menyelesaikan bagian-bagian kecil tersebut dasn terakhir menggabungkannya kembali (divide and conquer). Kompleksitas algoritma ini berada di antara $O(n^2)$ dan $O(n)$. Contoh algoritma yang memiliki kompleksitas ini adalah merge sort.



Gambar 7 Algoritma Merge Sort (Sumber : https://www.101computing.net/merge-sort-algorithm/)

5. $O(n^2)$

Algoritma yang memiliki kompleksitas ini hanya praktis digunakan untuk persoalan yang relatif kecil. Umumnya algoritma ini memproses perhitungan dalam 2 *nesting loop*. Contoh algoritma yang mempunyai kompleksitas ini adalah penjumlahan 2 matriks persegi NxN.

```
for (int i = 0 ; i < N; ++i){
    for (int j = 0; j < N; ++j){
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

Gambar 8 Penjumlahan matriks a dan b yang disimpan dalam matriks c (Sumber: koleksi pribadi)

6. $O(n^3)$

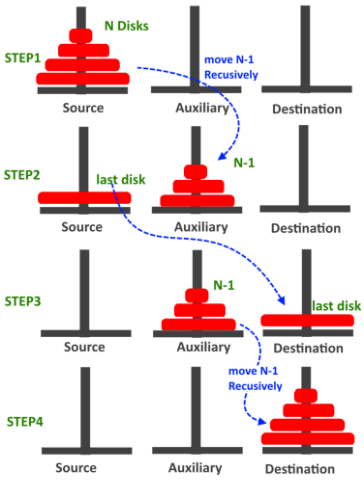
Seperti kompleksitas $O(n^2)$, algoritma yang memiliki kompleksitas ini memproses perhitungan dalam 3 *nesting loop*. Contoh algoritma yang memiliki kompleksitas ini adalah perkalian 2 matriks persegi NxN.

```
for (int i = 0 ; i < N; ++i){
    for (int j = 0; j < N; ++j){
        c[i][j] = 0;
        for (int k = 0; k < N; ++k){
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
        }
    }
}
```

Gambar 9 Perkalian matriks a dan b yang disimpan dalam matriks c (Sumber: koleksi pribadi)

7. $O(2^n)$

Algoritma yang memiliki kompleksitas ini biasanya tergolong kelompok algoritma yang mencari solusi persoalan secara *brute force*. Ada pula beberapa algoritma rekursif yang menghitung/memanggil hal yang sama hingga berulang kali. Contoh algoritma yang memiliki kompleksitas ini adalah penggeseran tiap piringan di permasalahan *Tower of Hanoi*.



Gambar 10 Solusi dari masalah Tower of Hanoi (Sumber https://medium.com/@jamalmaria111/tower-of-hanoi-js-algorithm-3f667fa46f0f)

8. $O(n!)$

Algoritma yang memiliki kompleksitas ini biasanya memproses tiap masukan dan menghubungkannya dengan n-1 masukan lainnya. Contoh algoritma yang memiliki kompleksitas ini adalah ekspansi kofaktor untuk menentukan determinan matriks NxN.

```
def determinan(M):
    # basis ketika matriks 2x2
    if len(M) == 2:
        return (M[0][0] * M[1][1]) - (M[0][1] * M[1][0])
    else:
        total = 0
        for kolom, elemen in enumerate(M[0]):
            # mengeluarkan baris pertama dan kolom sekarang
            K = [x[:kolom] + x[kolom + 1 :] for x in M[1:]]
            # jika kolom ganjil maka signnya minus
            if kolom % 2 == 0:
                total += elemen * determinan(K)
            else:
                total -= elemen * determinan(K)
        return total
```

Gambar 11 Mencari determinan menggunakan ekspansi kofaktor (Sumber: https://en.wikipedia.org/wiki/Laplace_expansion dengan sedikit pengubahan)

Dynamic programming adalah suatu metode untuk menyelesaikan masalah dengan cara membagi-bagi suatu permasalahan kompleks dengan memecahkannya menjadi bagian-bagian kecil lalu menyelesaikan masalah-masalah kecil tersebut dan kadang-kadang solusinya disimpan ke struktur data, misal array, map, set, dan berbagai struktur data lainnya (Coding Freak, 2018).

Ada dua pola penyelesaian menggunakan *dynamic programming*, yaitu dengan metode tabulasi dan memoisasi. Pada dasarnya metode tabulasi merupakan metode dari bawah ke atas (*bottom up*), sedangkan metode memoisasi merupakan metode dari atas ke bawah (*top down*). Perbedaannya bisa dilihat pada contoh berikut:

Versi *bottom up* : Saya akan belajar matematika diskrit dengan giat, lalu saya akan berlatih mengerjakan soal-soalnya hingga akhirnya saya akan menguasai mata kuliah ini.

Versi *top down* : Untuk menguasai mata kuliah matematika diskrit, saya harus berlatih mengerjakan soal-soalnya. Langkah pertama yang harus saya ambil adalah saya akan belajar matematika diskrit dengan giat.

Kedua metode di atas mempunyai kelebihan dan kekurangannya masing-masing, misal metode tabulasi lebih cepat prosesnya, tetapi relasi antarkondisi agak susah ditemukan di beberapa permasalahan. Sedangkan metode memoisasi lebih mudah menemukan relasi antarkondisinya, tetapi proses komputasinya kalah cepat dengan metode tabulasi karena masih mengandung rekursif.

III. BEBERAPA ALGORITMA SOLUSI BARISAN FIBONACCI DAN KOMPLEKSITAS ALGORITMANYA

Barisan fibonacci adalah barisan rekursif yang mempunyai relasi rekurens $f_n = f_{n-1} + f_{n-2}$, $f_0 = 0$ dan $f_1 = 1$. Ada banyak sekali algoritma yang dapat dipakai untuk mencari suku ke-n dari barisan fibonacci ini.

Salah satu algoritma yang mudah dipakai adalah dengan memanfaatkan relasi rekurens barisan fibonacci dan mencarinya dengan rekursif.

```
unsigned long long F(int n){
    if (n == 0 || n == 1) return n;
    else return F(n-1) + F(n-2);
}
```

Gambar 12 Fungsi suku ke-n barisan fibonacci secara rekursif (Sumber: koleksi pribadi)

Namun, belum tentu algoritma ini efisien. Mari kita hitung time complexity dari algoritma ini.

Kita punya $T(n) = T(n-1) + T(n-2)$ dari menghitung banyaknya operasi penjumlahan, dengan menganggap $T(n-1) \sim T(n-2)$ maka kita bisa memperoleh batas bawah, yaitu

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &= 2 * T(n-2) \\ &= 2^2 * T(n-4) \\ &= 2^3 * T(n-6) \end{aligned}$$

Didapat $T(n) = 2^k * T(n-2k)$, dan kita tahu bahwa $T(0) = 1$, maka jika kita mengambil k = n/2, kita akan mendapatkan $T(n) = 2^{n/2}$.

Selanjutnya kita dapat mencari batas atas dari kompleksitas algoritma ini, dengan menganggap $T(n-2) \sim T(n-1)$ kita bisa mendapatkan

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &= 2 * T(n-1) \\ &= 2^2 * T(n-2) \\ &= 2^3 * T(n-3) \end{aligned}$$

Didapat $T(n) = 2^k * T(n-k)$, dan kita tahu bahwa $T(0) = 1$, maka jika kita mengambil k = n, kita akan mendapatkan $T(n) = 2^n$. Jadi $2^{n/2} \leq T(n) \leq 2^n$, dan kita punya $T(n) = O(2^n)$. Jelas kompleksitas algoritma ini sangat buruk. Mari kita coba fungsi di atas dan hitung berapa waktu yang diperlukan untuk n=39, dan n= 50.

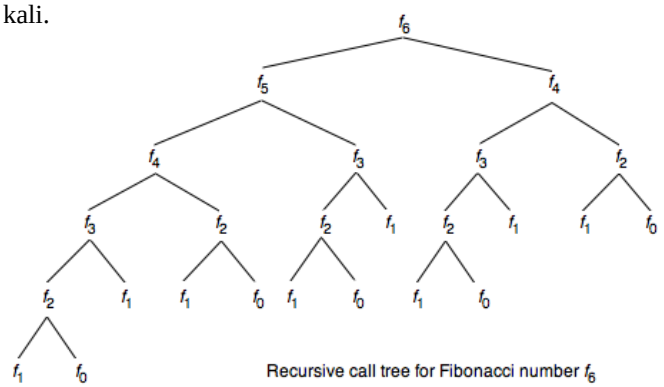
```
39
Bilangan fibonaccinya adalah 63245986
Waktu yang dibutuhkan untuk mencari adalah 1.028739 detik
```

Gambar 13 Fungsi fibonacci secara rekursif menerima inputan n = 39 (Sumber: koleksi pribadi)

```
50
Bilangan fibonaccinya adalah 12586269025
Waktu yang dibutuhkan untuk mencari adalah 210.902739 detik
```

Gambar 14 Fungsi fibonacci secara rekursif menerima inputan n = 50 (Sumber: koleksi pribadi)

Dari 2 gambar di atas, waktu yang dibutuhkan program meningkat drastis ketika masukannya berubah dari n = 39 menjadi n = 50. Hal ini disebabkan karena ketika memanggil fungsi rekursi, program akan mencari fungsi tersebut berulang kali.



Gambar 15 Pohon rekursif dari pencarian bilangan fibonacci ke 6 (Sumber: http://www.cse.unsw.edu.au/~billw/dictionaries/prolog/memoisation.html)

Dari pohon pemanggilan fungsi rekursif di atas, bisa kita lihat untuk menghitung f_6 program akan menghitung f_5 dan f_4 terlebih dahulu. Begitu selesai menghitung f_5, program akan menghitung lagi f_4, padahal f_4 sudah pernah dihitung sebelumnya ketika menghitung f_5. Hal ini menjadikan program tidak optimal, untuk mengoptimalkan kerja program, kita bisa mengaplikasikan *dynamic programming*.

Metode *dynamic programming* pertama yang kita gunakan untuk mengoptimalkan solusi barisan fibonacci adalah metode memoisasi atau metode *top down*. Metode ini masih menggunakan sifat rekursif, tetapi setiap memanggil fungsi fibonacci, hasil yang didapat langsung disimpan dalam array global. Jadi ide algoritmanya adalah kita mendeklarasikan array global, lalu mengubah seluruh elemennya menjadi nilai dumi tertentu. Jika sudah pernah dihitung, nilai dumi tersebut pasti pernah ditimpa dengan nilai bilangan fibonacci, sehingga program tidak perlu menghitung lagi. Namun, jika ternyata nilainya belum berubah, maka program akan menghitung secara rekursif. Lebih lengkapnya pada gambar potongan kode di bawah ini.

```
unsigned long long F[94];

unsigned long long Fibo(int n){
    if (F[n] != -1) return F[n];
    else{
        unsigned long long result;
        result = Fibo(n-1) + Fibo(n-2);
        F[n] = result;
        return result;
    }
}
```

Gambar 16 Fungsi fibonacci dengan memoisasi (Sumber: koleksi pribadi)

Sebelum memanggil fungsi Fibo(), dilakukan inisialisasi dengan mengisi seluruh elemen F dengan -1, lalu inisialisasi juga basis dari barisan fibonacci, yaitu dengan mengisi F[0]

dengan 0 dan F[1] dengan 1. Mari kita coba algoritma di atas dengan masukan n = 93. (Trivia: bilangan fibonacci ke 93 merupakan bilangan fibonacci terbesar yang dapat ditampung dalam unsigned long long di C, bilangan positif terbesar yang dapat ditampung di C adalah 2^{64}-1 = 18446744073709551615.)

```
93
Bilangan fibonaccinya adalah 12200160415121876738
Waktu yang dibutuhkan untuk mencari adalah 0.000043 detik
```

Gambar 17 Fungsi dengan memoisasi menerima masukan n = 93 (Sumber: koleksi pribadi)

Ternyata memoisasi mengoptimalkan waktu pencarian bilangan fibonacci dengan luar biasa. Mari kita hitung berapa kompleksitas algoritma ini. Kita masih mempunyai $T(n) = T(n-1) + T(n-2)$. Namun, karena bilangan fibonacci ke-(n-2) sudah dihitung dan dimasukkan ke cache ketika menghitung bilangan fibonacci ke-(n-1), maka waktu yang dibutuhkan untuk mengambil bilangan fibonacci ke-(n-2) adalah 1. Jadi kita punya $T(n) = T(n-1) + 1 = T(n-2) + 2 = \ldots = T(0) + n$ sehingga $T(n) = 1 + n$ (karena $T(0)$ adalah 1). Jadi kompleksitas algoritmanya adalah $T(n) = O(n)$ yang jelas jauh lebih baik daripada algoritma fungsi rekursif tanpa memoisasi.

Metode *dynamic programming* yang lain, metode tabulasi, juga mampu mengoptimalisasi waktu yang dibutuhkan program untuk mencari bilangan fibonacci ke-n. Berikut potongan kodenya.

```
unsigned long long Fibo(int n){
    if (n==0) return 0;
    if (n == 1) return 1;
    unsigned long long a,b,c;
    a= 0; b=1; c = 1;
    for (int i = 2; i <=n; ++i){
        c = a + b;
        a = c - a;
        b = c;
    }
    return c;
}
```

Gambar 18 Fungsi fibonacci dengan tabulasi (Sumber: koleksi pribadi)

Dengan metode tabulasi, kita dapat menghindari penggunaan memori berlebih pada metode memoisasi. Mari kita coba melihat waktu yang dibutuhkan untuk masukan n = 93 lagi.

```
93
Bilangan fibonaccinya adalah 12200160415121876738
Waktu yang dibutuhkan untuk mencari adalah 0.000026 detik
```

Gambar 19 Fungsi dengan tabulasi menerima masukan n = 93 (Sumber: koleksi pribadi)

Ternyata waktu yang dibutuhkan tidak terlalu berbeda jauh. Hal ini disebabkan karena metode tabulasi juga mempunyai kompleksitas $O(n)$.

Metode *bottom up* juga bisa memanfaatkan perkalian matriks 2x2. Ketika kita memangkatkan matriks A = {(1,1),(1,0)} sebanyak n kali, maka elemen A[0][0] berisi suku ke-n bilangan fibonacci, hal ini dapat dibuktikan melalui induksi matematika (tetapi tidak dibuktikan disini). Selain dengan induksi, ternyata perkalian dengan matriks A merupakan representasi dari algoritma dalam for loop di gambar 18. Sehingga ketika matriks A dipangkatkan sama saja dengan melakukan for loop pada algoritma yang direpresentasikan matriks A.

```
for (int i = 2; i <=n; ++i)
    c = a + b;
    a = c - a;
    b = c;
}
```

Gambar 19 Algoritma yang ditandai dapat direpresentasikan dengan perkalian matriks A (Sumber: koleksi pribadi)

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

@TamasGorbe

Gambar 20 Perpangkatan matriks A menghasilkan bilangan fibonacci (Sumber: https://twitter.com/tamasgorbe/status/1259143589772345344)

Ide algoritmanya sama dengan metode tabulasi sebelumnya. Namun, kompleksitasnya bisa diubah menjadi lebih cepat jika kita menggunakan fungsi perpangkatan matriks yang mirip dengan gambar 5. Kompleksitas algoritmanya berubah dari O(n) menjadi O(log n). Berikut potongan kodenya.

```
void power(unsigned long long F[2][2], int n){
    if (n==0 || n ==  1) return;
    unsigned long long A[2][2] = {{1,1},{1,0}};
    power(F,n/2);
    multiply(F,F);
    if (n % 2 != 0) multiply(F,A);
}

unsigned long long Fibo(int n){
    unsigned long long F[2][2] = {{1,1},{1,0}};
    if (n == 0) return 0;
    power(F, n-1);
    return F[0][0];
}
```

Gambar 21 Fungsi suku ke-n barisan fibonacci dengan matriks (Sumber: koleksi pribadi)

Fungsi multiply(X,Y) melakukan perkalian matriks biasa dan hasilnya disimpan dalam X. Sehingga algoritmanya memang mirip dengan algoritma dalam gambar 5. Mari kita coba untuk n = 93.

```
93
Bilangan fibonaccinya adalah 12200160415121876738
Waktu yang dibutuhkan untuk mencari adalah 0.000030 detik
```

Gambar 22 Fungsi fibonacci dengan matriks menerima n = 93 (Sumber: koleksi pribadi)

Ternyata waktunya tidak terlalu jauh dibandingkan tabulasi. Hal ini disebabkan masukannya relatif kecil, sehingga waktu yang dibutuhkan relatif sama.

Algoritma terakhir yang bisa digunakan adalah menggunakan solusi rekurens dari relasi rekurens barisan fibonacci. Kita punya :

$$f_n = f_{n-1} + f_{n-2}$$

Maka persamaan karakteristik dari barisan fibonacci adalah

$$r^2 - r - 1 = 0$$

sehingga $r_1 = \frac{1+\sqrt{5}}{2}$ dan $r_2 = \frac{1-\sqrt{5}}{2}$.

Dan diperoleh

$$f_n = \alpha_1 r_1^n + \alpha_2 r_2^n$$

Lalu, untuk $f_0 = 0$, didapat

$$0 = \alpha_1 + \alpha_2$$

sedangkan untuk $f_1 = 1$, didapat

$$1 = \alpha_1 r_1 + \alpha_2 r_2$$

Dari 2 persamaan terakhir, didapatkan $\alpha_1 = \frac{\sqrt{5}}{5}$ dan $\alpha_2 = -\frac{\sqrt{5}}{5}$.

Sehingga kita punya

$$f_n = \frac{\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Persamaan terakhir juga disebut *Binet's Formula.* Formula ini dapat langsung kita gunakan untuk mencari suku ke-n fibonacci. Berikut implementasinya dalam bahasa C

```
unsigned long long Fibo(int n){
    double alpha1,alpha2;
    alpha1 = sqrt(5)/5;
    alpha2 = alpha1*-1;
    double r1,r2;
    r1 = (1+ sqrt(5))/2;
    r2 = (1-sqrt(5))/2;
    return (alpha1*pow(r1,n) + alpha2*pow(r2,n));
}
```

Gambar 23 Fungsi fibonacci dengan menggunakan solusi rekurens (Sumber: koleksi pribadi)

Kompleksitas algoritma ini $O(1)$ alias konstan. Karena fungsi pow() yang berasal dari math.h memiliki kompleksitas algoritma yang konstan juga (berasal dari kode assembly FYL2X yang menghitung $y * \log_2 x$ lalu memangkatkan 2 dengan hasilnya menggunakan instruksi F2XM1. Sehingga diperoleh x^y dengan kompleksitas algoritma yang konstan). Namun algoritma ini memiliki kelemahan, yaitu komputer tidak bisa menampung bilangan irasional $\sqrt{5}$ secara utuh (hanya mendekati $\sqrt{5}$), sehingga ketika dipangkatkan dengan n yang cukup besar, errornya memiliki efek yang signifikan. Masukan terbesar yang masih tepat hasilnya adalah n = 71. Hasilnya mulai kurang tepat ketika n = 72.

```
71
Bilangan fibonaccinya adalah 308061521170129
Waktu yang dibutuhkan untuk mencari adalah 0.000112 detik
wiwid@wiwid-HP-Pavilion-g4-Notebook-PC:~/Desktop/kuliah/Se
71
Bilangan fibonaccinya adalah 308061521170129
Waktu yang dibutuhkan untuk mencari adalah 0.000047 detik
```

Gambar 24 Fungsi fibonacci dengan formula (atas) dan tabulasi (bawah) ketika menerima n = 71 (Sumber: koleksi pribadi)

```
72
Bilangan fibonaccinya adalah 498454011879265
Waktu yang dibutuhkan untuk mencari adalah 0.000127 detik
wiwid@wiwid-HP-Pavilion-g4-Notebook-PC:~/Desktop/kuliah/S
72
Bilangan fibonaccinya adalah 498454011879264
Waktu yang dibutuhkan untuk mencari adalah 0.000068 detik
```

Gambar 25 Fungsi fibonacci dengan formula (atas) dan tabulasi (bawah) ketika menerima n = 72 (Sumber: koleksi pribadi)

Jadi untuk n jauh lebih besar dari 71, semakin besar pula error yang didapat. Oleh karena itu, algoritma ini lebih cocok digunakan ketika n ≤ 71.

IV. KESIMPULAN

Setiap persoalan yang dihadapi *programmer* pasti memiliki banyak penyelesaian dengan pendekatan yang berbeda-beda. Namun, tidak semua cara penyelesaiannya efisien. Kompleksitas algoritma menjadi pegangan penting bagi *programmer* untuk memilih algoritma mana yang cocok digunakan. Seperti halnya persoalan barisan fibonacci yang memiliki berbagai algoritma penyelesaian dengan kompleksitas algoritma yang bermacam-macam, mulai dari $O(2^n)$ hingga $O(1)$. *Programmer* harus mampu menyusun strategi algoritma yang sesuai, bisa dengan mengaplikasikan *dynamic programming*, *greedy*, atau strategi algoritma lain yang sesuai. Tidak lupa untuk memperhatikan batas atas dari masukan sehingga *programmer* cukup membuat penyelesaian yang sudah cukup untuk masukan dengan nilai batas terbesar. Misal pada makalah ini, penulis ingin menyelesaikan permasalahan barisan fibonacci dengan menggunakan bahasa C yang hanya bisa menyimpan bilangan fibonacci ke-93 (tentu bahasa pemrograman lain, misalnya python, memiliki batas yang berbeda. Penulis menggunakan C karena kecepatan programnya). Sehingga andai penulis ingin memilih algoritma mana yang sesuai, penulis cukup memilih algoritma yang memiliki kompleksitas $O(n)$ karena batas atas dari n, yaitu 93, masih relatif kecil.

V. UCAPAN TERIMA KASIH

Penulis bersyukur kepada Tuhan Yang Maha Esa karena atas berkat-Nya, penulis mampu menyelesaikan makalah berjudul "Berbagai Kompleksitas Algoritma Pencari Suku ke-N Barisan Fibonacci dari Rekursif, *Dynamic Programming*, hingga Solusi Rekurens". Penulis mengucapkan terima kasih kepada orang tua, sahabat-sahabat, dan teman-teman penulis yang setia memberikan dukungan, motivasi, dan masukan terhadap penyelesaian tugas pembuatan makalah ini .Penulis juga berterima kasih kepada Bapak Dr. Ir. Rinaldi Munir, M.T., dosen mata kuliah IF2120 Matematika Diskrit pada semester I 2020/2021, atas ilmu yang sudah diberikan pada penulis.

REFERENSI

[1] Munir, Rinaldi. 2020. "Rekursi dan Relasi Rekurens (Bagian 1)". http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Rekursi-dan-relasi-rekurens-(Bagian-1).pdf (Diakses pada tanggal 7 Desember 2020 Pukul 15.00 WIB)

[2] Munir, Rinaldi. 2020. "Rekursi dan Relasi Rekurens (Bagian 2)". http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Rekursi-dan-relasi-rekurens-(Bagian-2).pdf (Diakses pada tanggal 7 Desember 2020 Pukul 15.00 WIB)

[3] Freak, Coding. 2018. "Top 50 Dynamic Programming Practice Problems". https://blog.usejournal.com/top-50-

dynamic-programming-practice-problems-4208fed71aa3 (Diakses pada tanggal 7 Desember 2020 Pukul 15.00 WIB)
[4] Geeksforgeeks. 2019. “Tabulation vs Memoization”. https://www.geeksforgeeks.org/tabulation-vs-memoization/ (Diakses pada tanggal 7 Desember 2020 Pukul 15.00 WIB)
[5] Ahmed, Syed Tousif. 2018. “Fibonacci Iterative vs Recursive”. https://medium.com/@syedtousifahmed/fibonacci-iterative-vs-recursive-5182d7783055 (Diakses pada tanggal 7 Desember 2020 Pukul 15.00 WIB)
[6] ICS 161. 1996. “Design and Analysis of Algorithms” https://www.ics.uci.edu/~eppstein/161/960109.html (Diakses pada tanggal 7 Desember 2020 Pukul 15.00 WIB)
[7] Geeksforgeeks. 2020. “Program for Fibonacci numbers” https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/ (Diakses pada tanggal 7 Desember 2020 Pukul 15.00 WIB)

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, 10 Desember 2020

Widya Anugrah Putra 13519105