Algoritma Pencarian Bilangan Fibonacci

Daniel Pintara - 13515071[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*13515071@std.stei.itb.ac.id*
nieltansah@gmail.com

***Abstrak*— Kemangkusan algoritma merupakan salah satu faktor penting yang mempengaruhi kebutuhan waktu dari suatu algoritma. Jika algoritma tidak mangkus, maka pertambahan kecepatan pada komputer juga tidak memberikan dampak besar. Metode yang sering digunakan untuk menentukan kemangkusan algoritma adalah mencari notasi O besar. Pada makalah ini, penulis membandingkan beberapa algoritma mencari bilangan Fibonacci dari yang tidak mangkus sampai yang mangkus, dan membandingkan kecepatan dari algoritma-algoritma itu.**

***Kata Kunci*— efisiensi algoritma, Fibonacci, kompleksitas algoritma, notasi O besar.**

I. Pendahuluan

Kemangkusan algoritma merupakan salah satu faktor penting yang mempengaruhi kebutuhan waktu dari suatu algoritma. Jika algoritma tidak mangkus, maka pertambahan kecepatan pada komputer tidak mengubah lama waktu yang dibutuhkan algoritma tersebut untuk menyelesaikan suatu persoalan secara signifikan.

Pertambahan kecepatan pada komputer dapat dianalogikan sebagai pertambahan kecepatan dari operasi-operasi elementer, seperti penjumlahan, perkalian, dan sebagainya, sedangkan, algoritma itu sendiri dapat dianalogikan sebagai strategi dalam menyelesaikan suatu persoalan. Jika waktu yang dibutuhkan untuk melakukan operasi-operasi elementer itu dipersingkat, maka waktu yang dibutuhkan bagi algoritma tersebut untuk menyelesaikan suatu persoalan menjadi lebih singkat. Namun, hal itu tidak cukup. Saat terjadi pertambahan beban persoalan, jumlah operasi-operasi elementer yang dilaksanakan untuk menyelesaikan persoalan itu juga mengalami pertambahan yang signifikan pada algoritma yang tidak mangkus, sehingga waktu komputasi yang dibutuhkan untuk menyelesaikan persoalan itu kembali menjadi besar. Hal ini tidak sejalan dengan upaya yang dibutuhkan untuk membeli suatu komputer yang lebih cepat.

Dari paragraf di atas dapat disimpulkan bahwa komputer yang cepat belum cukup untuk mengatasi persoalan-persoalan yang ada, namun, strategi untuk menyelesaikan persoalan-persoalan itu juga harus dibenahi. Jika suatu algoritma yang mangkus dilaksanakan pada komputer yang lebih cepat dan terjadi pertambahan beban persoalan, memang ada pertambahan pada jumlah operasi-operasi elementer yang dilaksanakan untuk menyelesaikan persoalan itu, namun, pertambahan itu tidak signifikan pada algoritma yang mangkus, sehingga waktu komputasi yang dibutuhkan tetap kecil.

Untuk menentukan seberapa mangkus suatu algoritma, diperlukan suatu metode untuk mengetahuinya. Selisih waktu dari proses komputasi tidak digunakan untuk menentukan seberapa mangkus suatu algoritma karena hal itu dipengaruhi oleh beberapa faktor seperti jenis komputer, sistem operasi, dan lain-lain, sehingga perhitungan yang dihasilkan adalah perhitungan yang tidak bersifat universal. Metode yang sering dipakai untuk menentukan seberapa mangkus suatu algoritma adalah mencari nilai notasi O besar dengan cara menganalisa algoritma itu.

Namun, seberapa besar perbedaan waktu komputasi yang dibutuhkan untuk menyelesaikan suatu persoalan dengan algoritma yang mangkus dan algoritma yang kurang mangkus? Oleh karena itu, penulis akan membahas beberapa algoritma yang digunakan untuk mencari bilangan fibonacci. Bilangan fibonacci itu sendiri adalah bilangan yang sering ditemukan di alam, contohnya adalah susunan daun, sarang lebah, dan lain-lain. Mencari bilangan fibonacci merupakan suatu persoalan yang cukup menantang, karena untuk menemukan strategi yang lebih cepat untuk mencari bilangan fibonacci, diperlukan pemahaman akan teori bilangan, khususnya sifat-sifat dari bilangan fibonacci. Penulis membandingkan beberapa algoritma fibonacci dari yang kurang mangkus, sampai yang menurut penulis cukup mangkus, sehingga pembaca dapat menilai sendiri berapa pengaruh kemangkusan dari suatu algoritma itu sendiri terhadap waktu yang dibutuhkan untuk menyelesaikan suatu persoalan.

II. Landasan Teori

A. Kompleksitas Algoritma

Algoritma merupakan suatu cara yang sistematis untuk menyelesaikan suatu persoalan. Algoritma tidak saja harus menghasilkan hasil yang tepat, namun algoritma juga harus mangkus. Algoritma yang benar, namun tidak mangkus, mungkin tidak berguna untuk menyelesaikan suatu persoalan dengan beban yang berat.

Jumlah operasi-operasi elementer yang diperlukan untuk menyelesaikan suatu persoalan dan kebutuhan memori bertambah secara signifikan pada algoritma yang tidak mangkus, dan menyebabkan komputer menjadi tidak

mampu untuk menangani persoalan tersebut.

a. Kompleksitas Waktu

Seperti yang disinggung sebelumnya, selisih waktu merupakan suatu hal yang kurang tepat jika digunakan untuk mengetahui seberapa mangkus suatu algoritma, karena banyak faktor-faktor lain yang mempengaruhi hasil dari perbandingan waktu itu, seperti jenis mesin, sistem operasi, dan sebagainya. Oleh karena itu, dibutuhkan suatu metode untuk mengetahui seberapa mangkus suatu algoritma.

Metode yang sering digunakan untuk mengetahui seberapa mangkus dari suatu algoritma adalah metode Analisa algoritma yang menghasilkan nilai notasi O besar. Notasi ini nantinya dapat dibandingkan dengan nilai notasi O besar dari algoritma lainnya untuk mengetahui algoritma apa yang lebih mangkus.

Perhitungan kompleksitas algoritma diawali dengan menghitung banyaknya operasi-operasi dasar yang dilaksanakan. Suatu algoritma mungkin terdiri dari berbagai jenis operasi-operasi dasar, seperti operasi penjumlahan, operasi pengurangan, operasi pengisian nilai, operasi perbandingan, operasi pembagian, operasi pembacaan, dan sebagainya. Jika waktu untuk menyelesaikan operasi-operasi dasar tersebut diketahui, maka waktu yang dibutuhkan untuk menyelesaikan persoalan itu juga dapat diketahui.

Sebagai contoh, perhatikan algoritma di bawah ini:

```
procedure HitungRerata
    (input  a : array[1..n] of integer,
     output r : real)
{ Menghitung rata-rata dari array a }
KAMUS
    k : integer
    j : real
ALGORITMA
    j ← 0
    k ← 1

    while k ≤ n do
        j ← j + ak
        k ← k + 1

    { k > n }
    r ← j / n
```

Berikut ini adalah jenis-jenis operasi dasar yang digunakan untuk menyelesaikan persoalan mencari rata-rata pada sebuah larik:

- operasi pengisian nilai (operator ←)
- operasi penjumlahan (operator +)
- operasi pembagian (operator /)

Kompleksitas algoritma dapat dihitung dengan cara menghitung masing-masing jumlah operasi dasar yang dilaksanakan. Jika operasi tersebut ada di dalam kalang, maka jumlah operasi dasar tersebut bergantung pada berapa kali kalang tersebut mengalami pengulangan.

(i) Operasi Pengisian Nilai

Jumlah seluruh operasi pengisian nilai:

$t_1 = 1 + 1 + n + n + 1 = 3 + 2n$

(ii) Operasi Penjumlahan

Jumlah seluruh operasi penjumlahan:

$t_2 = n + n = 2n$

(iii) Operasi Pembagian

Jumlah seluruh operasi pembagian:

$t_3 = 1$

Jika satu kali operasi pengisian nilai membutuhkan waktu selama a detik, satu kalo operasi penjumlahan membutuhkan waktu selama b detik, dan satu kali operasi pembagian membutuhkan waktu selama c detik, kebutuhan waktu sesungguhnya dapat dihitung dengan cara:

$t = t_1 + t_2 + t_3 = (3 + 2n)a + 2nb + c$

Namun, model perhitungan di atas kurang dapat diterima karena tidak ada informasi yang tepat mengenai berapa lama suatu operasi dasar dilaksanakan. Oleh karena itu, kompleksitas waktu diukur sebagai fungsi $T(n)$ yang menghubungkan antara bobot persoalan n dengan jumlah operasi-operasi dasar yang dibutuhkan untuk menjalankan suatu algoritma.

Idealnya, semua operasi dasar pada algoritma harus dianalisa, namun untuk alasan praktis, hanya operasi dasar yang mendasari, atau dominan pada suatu algoritma saja yang dianalisa.

Sebagai contoh, perhatikan algoritma di bawah ini:

```
procedure CariElemenTerbesar
    (input  a : array[1..n] of integer,
     output m : integer)
{ Mencari nilai terbesar dari array a }
KAMUS
    k : integer
ALGORITMA
    m ← a1
    k ← 2

    while k ≤ n do
        if ak < m then
            m ← ak
        i ← i + 1
    { k > n }
```

Pada kasus di atas, operasi dasar yang dominan atau mendasari algoritma pencarian nilai terbesar adalah operasi perbandingan, oleh karena itu operasi itulah yang akan dianalisa untuk mendapatkan kompleksitas keseluruhan dari algoritma itu.

Operasi perbandingan pada kasus di atas terletak dalam sebuah kalang yang diulang sebanyak $n - 1$ kali. Oleh karena itu, kompleksitas dari algoritma pencarian elemen terbesar:

$T(n) = n + 1$

Hal lain yang harus diperhatikan untuk menentukan kompleksitas algoritma adalah karakteristik dari masukan yang diberikan. Ada algoritma tertentu yang bergantung pada beberapa faktor, seperti algoritma pencarian nilai pada sebuah larik yang tidak hanya bergantung pada jumlah elemen `n` pada larik, namun juga elemen `x` yang dicari.

Kompleksitas waktu dibedakan menjadi tiga, yaitu:

1. $T_{max}(n)$: kompleksitas waktu untuk kasus terburuk
2. $T_{min}(n)$: kompleksitas waktu untuk kasus terbaik
3. $T_{avg}(n)$: kompleksitas waktu untuk waktu rata-rata

Sebagai contoh, perhatikan algoritma di bawah ini:

```
procedure PencarianBeruntun
    (input  a : array[1..n] of integer,
     input  x : integer,
     output i : integer)
{ Mencari index elemen x pada array a }
KAMUS
    k : integer
    f : boolean
ALGORITMA
    k ← 1
    f ← false

    while (k ≤ n) and (not f) do
        if (ak = x) then
            f ← true
        else
            k ← k + 1
    { k > n atau ketemu }

    if f then
        i ← k
    else
        i ← 0
```

Jika diperhatikan, algoritma pencarian beruntun membandingkan setiap elemen larik dengan elemen `x`. Jika elemen `x` ditemukan, maka proses pencarian akan dihentikan. Oleh karena itu:

1. Kasus Terbaik (*best case*)
   Kasus terbaik terjadi saat elemen x ada sebagai elemen pertama dari larik.

   $T_{min}(n) = 1$

2. Kasus Terburuk (*worst case*)
   Kasus terburuk terjadi saat elemen x ada sebagai elemen terakhir pada larik.

   $T_{max}(n) = n$

3. Kasus Rata-Rata (*average case*)
   Jika `x` ditemukan pada posisi ke-*j*, maka operasi perbandingan akan dilakukan sebanjak *j*-kali.

   $T_{avg}(n) = (n + 1) / 2$

Dalam perhitungan kompleksitas algoritma, yang menjadi menarik seringkali bukanlah kompleksitas waktu yang presisi, namun bagaimana waktu terbaik dan waktu terburuk tumbuh (meningkatkan jumlah operasi dasar) bersama-sama seiring dengan meningkatnya bobot persoalan yang ditangani suatu algoritma. Hal ini dikenal sebagai kompleksitas waktu asimtotik.

Kompleksitas waktu asimtotik terdiri dari 3 jenis, yaitu notasi O besar, notasi Θ besar, dan notasi Ω besar. Nilai dari ketiga notasi tersebut dapat ditentukan dari fungsi `T(n)` dengan cara menentukan orde `f(n)` yang sesuai dan suatu konstanta `C`.

Notasi O besar dapat ditentukan dengan mencari orde paling besar dari `f(n)` dan konstanta `C` sedemikian rupa sehingga:

$T(n) \leq C(f(n))$

Notasi Ω besar dapat ditentukan dengan mencari orde paling kecil dari `f(n)` dan konstanta `C` sedemikian rupa sehingga:

$T(n) \leq C(f(n))$

Notasi Θ besar dapat ditentukan dengan mencari orde yang sama dengan `h(n)` jika persamaan-persamaan ini dipenuhi:

$T(n) = O(h(n))$
$T(n) = \Omega(g(n))$

Berikut ini adalah kompleksitas waktu asimtotik yang umum dalam bentuk notasi O besar.

1. **O (1)**
   Pelaksanaan waktu dari algoritma tidak dipengaruhi oleh ukuran masukan.

   Algoritma dengan nilai notasi O besar ini ditemukan pada algoritma pengaksesan elemen pada larik, dan sebagainya.

2. **O (log n)**
   Pelaksanaan waktu dari algoritma lebih lambat dari pada laju pertumbuhan bobot persoalan.

   Algoritma dengan nilai notasi O besar ini ditemukan pada algoritma pencarian biner, dan sebagainya.

3. **O (n)**
   Pelaksanaan waktu dari algoritma berjalan seiring dengan laju pertumbuhan bobot persoalan.

   Algoritma dengan nilai notasi O besar ini ditemukan pada algoritma pencarian sequensial, dan sebagainya.

4. **O (n log n)**
   Algoritma dengan nilai notasi O besar ini sering ditemukan pada algoritma yang membagi persoalan menjadi persoalan yang lebih kecil, menyelesaikan persoalan secara independen, dan menggabungkan solusi dari masing-masing persoalan.

   Sebagai contoh, algoritma jenis itu ditemukan pada algoritma *merge sort*, dan sebagainya.

5. **O (n^2)**
   Algoritma ini hanya praktis dilaksanakan pada

persoalan dengan bobot yang ringan.

Algoritma dengan nilai notasi O besar ini ditemukan pada algoritma *selection sort* (pengurutan dengan memilih elemen untuk menggantikan elemen teratas), dan sebagainya.

6. **O (n^3)**
   Algoritma seperti ini menyelesaikan suatu persoalan dengan 3 kalang yang bersarang.

   Algoritma dengan nilai notasi O besar ini ditemukan pada algoritma perkalian matriks secara sederhana, dan sebagainya.

7. **O (2^n)**
   Algoritma yang tergolong dalam kelompok ini merupakan algoritma yang mencari solusi dengan menggunakan teknik *brute force*.

   Algoritma dengan nilai notasi O besar ini ditemukan pada algoritma pencarian sirkuit Hamilton.

8. **O (n!)**
   Algoritma jenis ini memproses setiap masukan dan menghubungkannya dengan `n - 1` masukan lainnya.

   Algoritma dengan nilai notasi O besar ini ditemukan pada algoritma untuk menyelesaikan *Travelling Salesperson Problem*.

b. Kompleksitas Ruang

Kompleksitas ruang adalah ukuran untuk mengukur banyak memori yang diperlukan untuk melaksanakan suatu algoritma. Ketika suatu komputer melaksanakan suatu algoritma yang memerlukan banyak memori, dan di suatu titik komputer tersebut tidak lagi memiliki memori kosong untuk algoritma tersebut, maka pemrosesan dari algoritma tersebut mungkin terhenti sehingga tidak menghasilkan suatu hasil akhir.

Kebutuhan memori mungkin masih dapat dipaksakan untuk tercukupi dengan memanfaatkan memori eksternal lainnya, seperti harddisk, dan sebagainya. Namun, penggunaan harddisk sebagai memori tambahan (pagefile) dapat membuat pemrosesan algoritma menjadi sangat lama, karena laju pembacaan dan penulisan pada harddisk jauh dibawah laju pembacaan dan penulisan pada RAM.

Kompleksitas ruang pada algoritma yang menyelesaikan suatu persoalan dengan pendekatan iteratif tidak menjadi masalah jika algoritma tersebut melakukan alokasi memori sebelum iterasi dimulai dan tidak memanggil fungsi lain yang memerlukan alokasi memori tambahan.

Sebagai contoh, perhatikan algoritma berikut ini:

```
procedure Tambah
    (input/output
            a : array[1..n] of integer,
     input   t : integer)
{ Menambah nilai t pada setiap elemen a }
KAMUS
    i : integer
ALGORITMA
    i ← 1
    while (i ≤ n) do
        aᵢ ← aᵢ + t
```

Algoritma ini mengalokasikan memori sebelum iterasi dimulai dan tidak memerlukan memori tambahan saat bekerja. Oleh karena itu, algoritma ini memiliki kompleksitas ruang `O(1)` dalam notasi O besar.

Kompleksitas ruang perlu diperhatikan pada algoritma yang menggunakan pendekatan rekursif untuk menyelesaikan suatu persoalan. Saat fungsi memanggil fungsi lain, maka memori yang dipakai pada fungsi pemanggil akan tetap tersimpan karena suatu saat fungsi yang dipanggil akan keluar, dan control program kembali pada fungsi pemanggil dengan nilai variabel-variabel yang tidak berubah (kecuali untuk variable yang referensinya diberikan pada fungsi yang dipanggil, atau variabel menyimpan hasil keluaran dari fungsi yang dipanggil). Memori baru akan dibebaskan ketika keluar dari fungsi.

B. Identitas Fibonacci oleh Donald E. Knuth

Ada beberapa cara untuk mencari nilai dari bilangan Fibonacci dari bilangan Fibonacci sebelumnya, salah satunya dengan identitas matrix bilangan Fibonacci yang ditemukan oleh Donald E. Knuth.

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

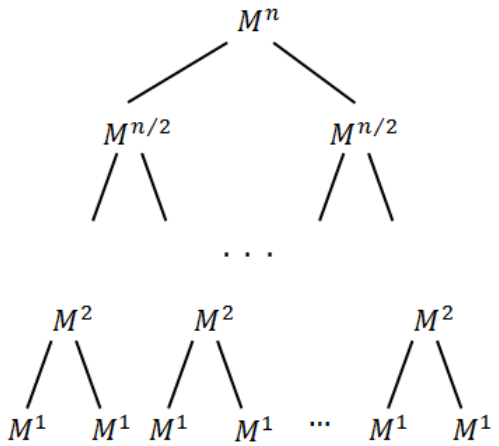
Identitas ini dapat dibuktikan menggunakan induksi matematika. Misal, terdapat sebuah matrix Q.

$$Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Sesuai dengan identitas:

$$F_n = Q^{n-1}\,(1,1)$$

Untuk menghitung F_n, maka nilai Q^{n-1} dikalkulasi terlebih dahulu. Perhitungan F_n merupakan pemangkatan dari matriks. Oleh karena itu, jika ada matriks `M` yang akan dipangkatkan, maka M^n akan membentuk pohon:



Jadi, untuk menghitung matriks M^n, maka nilai matriks

$M^{n/2}$ harus dikalkulasi terlebih dahulu dan dikalikan dengan dirinya sendiri, dan seterusnya. Tinggi dari pohon tersebut adalah `log n`, sehingga kompleksitas perkalian matriks itu adalah `O(log n)`.

Sebagai optimasi, rumus-rumus berikut ini dapat diturunkan dari konsep di atas. Jika diketahui `F(k)` dan `F(k+1)`, maka:

$$F(2k) = F(k)[2F(k+1) - F(k)]$$
$$F(2k+1) = F(k+1)^2 + F(k)^2$$

III. Analisis Kasus

A. Pendekatan Rekursif

Algoritma untuk mencari bilangan Fibonacci dengan menggunakan pendekatan rekursif merupakan salah satu algoritma yang cukup primitif.

Basis:

```
F(0) = 0
F(1) = 1
```

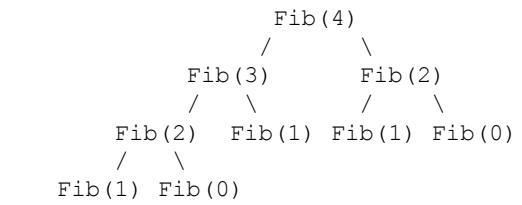
Rekurens:

```
F(n) = F(n-1) + F(n-2)
```

Dengan melihat rumus di atas, dengan mudah dapat disimpulkan bahwa untuk mencari bilangan Fibonacci `F(n)`, maka diperlukan nilai `F(n-1)` dan nilai `F(n-2)`. Berdasarkan pengetahuan sebelumnya, algoritma untuk mencari bilangan Fibonacci dapat dibuat dengan pendekatan rekursif.

```
function FibRekursif
    (input n : integer) : integer
{ Mencari bil. Fibonacci }
ALGORITMA
    if (n = 0) then
        → 0
    else
    if (n = 1) then
        → 1
    else
        → FibRekursif(n - 1)
        + FibRekursif(n - 2)
```

Jika diperhatikan, kode tersebut jika dieksekusi akan membentuk pohon eksekusi. Sebagai contoh, berikut ini adalah pohon eksekusi dari pencarian `F(4)`.



Relasi rekurens dari fungsi tersebut dapat dianalisa untuk menghitung nilai notasi O besar.

```
T(n<=1) = O(1)
T(n) = T(n-1) + T(n-2) + O(1)
```

Dari kedua rumus diatas, diketahui bahwa nilai notasi O besar pada algoritma tersebut adalah $O(2^n)$.

Sedangkan untuk kompleksitas ruang, algoritma ini merupakan algoritma dengan pendekatan rekursif, berarti memori akan dibebaskan saat keluar dari suatu fungsi. Oleh karena itu, kompleksitas ruang algoritma dapat dihitung dengan melihat tinggi dari pohon rekursi, karena di titik itu, semua variabel rekursi sedang disimpan.

Jika diperhatikan, aras kiri dari pohon rekursi Fibonacci merupakan aras dengan simpul tertinggi, karena di simpul itu, algoritma menghitung `F(n-1)` yang tentunya memerlukan proses rekursi yang lebih dalam daripada `F(n-2)`. Tinggi maksimum pohon adalah `n` karena saat itu algoritma sedang menghitung dari `n` sampai basis. Oleh karena itu, dapat disimpulkan bahwa kompleksitas ruang dari algoritma tersebut adalah `O(n)`.

Jika diperhatikan, algoritma seperti ini bukanlah algoritma yang mangkus. Hal ini dibuktikan dari waktu penyelesaian algoritma tersebut di komputer penulis dengan prosesor Intel® Processor 5Y10 @ 0.80Ghz. Penulis menulis program dengan bahasa C++. Program dijalankan pada system operasi Microsoft® Windows® 10 pada lingkungan *Bash on Ubuntu on Windows*.

Eksekusi `F(44)`:

```
$ time ( echo 44 | bin/fib3 )
701408733

real    0m7.749s
user    0m7.672s
sys     0m0.063s
```

Eksekusi `F(45)`:

```
$ time ( echo 45 | bin/fib3 )
1134903170

real    0m12.811s
user    0m12.469s
sys     0m0.078s
```

B. Pendekatan Iteratif

Selain pendekatan rekursif, pencarian bilangan Fibonacci dapat dilakukan dengan pendekatan iteratif. Keuntungan dari pendekatan ini adalah pendekatan ini tidak menggunakan memori tambahan setelah iterasi dimulai, sehingga kompleksitas ruang dari pendekatan ini adalah `O(1)`.

```
function FibIteratif
    (input n : integer) : integer
{ Mencari bil. Fibonacci }
KAMUS
    a, b : integer
    c, i : integer
ALGORITMA
    a ← 0
    b ← 1
    i iterate [0..(n - 1)]
        c ← a
        a ← b
```

```
        b ← c + b
    → a
```

Jika diperhatikan, algoritma ini melakukan iterasi sebanyak `n` kali. Oleh karena itu, dapat disimpulkan bahwa algoritma ini memiliki kompleksitas `O(n)`.

Algoritma ini lebih mangkus dari algoritma sebelumnya. Penulis dapat menghitung Fibonacci ke-1.000.000.000 modulus 2^{64} hanya dengan waktu 4 detik saja.

Eksekusi `F(1000000000)`:

```
$ time ( echo 1000000000 | bin/fib2 )
3311503426941990459

real    0m4.172s
user    0m4.125s
sys     0m0.047s
```

Eksekusi `F(2000000000)`:

```
$ time ( echo 2000000000 | bin/fib2 )
26977638455882275 25

real    0m8.297s
user    0m8.203s
sys     0m0.094s
```

C. Menggunakan Identitas Fibonacci

Selain pendekatan iteratif, pencarian bilangan Fibonacci juga dapat dilakukan dengan menggunakan kombinasi dari rumus Fibonacci ke-`n+1` dan rumus Fibonacci ke-`2n`. Penulis melakukan sedikit penelitian untuk menemukan cara mengkombinasikan rumus Fibonacci ke-`n+1` dan rumus Fibonacci ke-`2n` sehingga diperoleh suatu algoritma yang mangkus untuk keperluan *competitive programming* beberapa bulan yang lalu.

Kombinasi dengan rumus Fibonacci ke `n+1` diperlukan untuk menghitung nilai `n` yang tidak genap 2^n dengan cara memperhatikan *bit* dari `n`. Sebagai contoh, desimal 10 memiliki representasi biner 1010. Untuk mencari Fibonacci ke-10, maka algoritma akan memulainya dengan bilangan Fibonacci ke-1. *Bit* pertama dilewati, lalu, jika algoritma menemukan suatu *bit*, maka algoritma akan menghitung Fibonacci `2n`. Jika bit yang diproses merupakan 1, maka algoritma juga akan menghitung Fibonacci `n+1`.

`F(1)` menggunakan rumus Fibonacci `2n`, menjadi:
`F(2)` menggunakan rumus Fibonacci `2n`, menjadi:
`F(4)` menggunakan rumus Fibonacci `n+1`, menjadi:
`F(5)` menggunakan rumus Fibonacci `2n`, menjadi:
`F(10)` dan proses selesai.

Kasus terburuk terjadi jika terdapat banyak *bit* pada `n`, dan masing-masing dari *bit* itu bernilai 1. Berikut ini adalah contoh pemrosesan `F(15)` dimana `n` memiliki *bit* 1111.

`F(1)` menggunakan rumus Fibonacci `2n`, menjadi:
`F(2)` menggunakan rumus Fibonacci `n+1`, menjadi:
`F(3)` menggunakan rumus Fibonacci `2n`, menjadi:
`F(6)` menggunakan rumus Fibonacci `n+1`, menjadi:
`F(7)` menggunakan rumus Fibonacci `2n`, menjadi:
`F(14)` menggunakan rumus Fibonacci `n+1`, menjadi:
`F(15)` dan proses selesai.

Berikut ini adalah algoritma dari penjelasan di atas.

```
function FibCepat
    (input n : integer) : integer
{ Mencari bil. Fibonacci }
KAMUS
    a, b : integer
    c, d : integer
    mask : integer
    i : integer
ALGORITMA
    a ← 0
    b ← 1

    i iterate [jml_bit_terpakai(n)..0]
        c ← (b << 1) - a
        c ← c * a

        d ← b * b
        d ← d + (a * a)

        a ← c
        b ← d
        mask ← 1 << i

        if (n & mask) != 0 then
            c ← a
            a ← bil
            b ← a + c

    → a
```

Algoritma ini sendiri memiliki kompleksitas ruang sebesar `O(1)` karena menggunakan pendekatan iteratif saja. Selain itu, algoritma ini juga memiliki kompleksitas waktu sebesar `O(log n)`.

Melalui algoritma ini, penulis dapat menghitung bilangan Fibonacci ke-2^{64} modulus 2^{64} hanya dalam sekejap mata. Untuk mencari jumlah *bit* yang terpakai pada `n`, penulis menggunakan fungsi `__builtin_clzl` yang sudah tersedia menggunakan *compiler* `g++`.

```
$ time ( echo 18446744073709551616 |
bin/fib1 )
800812746651928290

real    0m0.047s
user    0m0.000s
sys     0m0.047s
```

IV. KESIMPULAN

Berikut ini adalah kesimpulan yang diambil berdasarkan analisis, percobaan, dan perhitungan yang telah dilakukan sebelumnya.

1. Kompleksitas algoritma merupakan faktor yang sangat mempengaruhi waktu yang diperlukan untuk menyelesaikan suatu persoalan.
2. Untuk merumuskan suatu algoritma yang mangkus, maka dibutuhkan penelitian dan pemahaman yang

baik terhadap persoalan yang dihadapi.

3. Laju pertumbuhan algoritma yang diformulasikan dalam bentuk notasi O besar tercermin pada waktu yang dibutuhkan algoritma untuk menyelesaikan suatu persoalan.

REFERENSI

[1] Munir, Rinaldi. 2006. Diktat Kuliah IF2120 Matematika Diskrit, Edisi Keempat. Informatika Bandung : Bandung.

[2] http://jwilson.coe.uga.edu/emat6680/parveen/fib_nature.htm
Tanggal akses: 9 Desember 2016, pukul 06:50 WIB

[3] https://www.nayuki.io/page/fast-fibonacci-algorithms
Tanggal akses: 9 Desember 2016, pukul 07:00 WIB

[4] http://kukuruku.co/hub/algorithms/the-nth-fibonacci-number-in-olog-n
Tanggal akses: 9 Desember 2016, pukul 07:02 WIB

[5] https://www.cs.northwestern.edu/academics/courses/311/html/space-complexity.html.
Tanggal akses: 9 Desember 2016, pukul 10:00 WIB

[6] http://stackoverflow.com/questions/360748/computational-complexity-of-fibonacci-sequence
Tanggal akses: 9 Desember 2016, pukul 11:10 WIB

[7] http://stackoverflow.com/questions/15047116/a-iterative-algorithm-for-fibonacci-numbers
Tanggal akses: 9 Desember 2016, pukul 11:56 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, 9 Desember 2016

Daniel Pintara
13515071