Penyelesaian Barisan Rekursif dengan Kompleksitas Logaritmik Menggunakan Pemangkatan Matriks

Luqman Arifin Siswanto - 13513024
Program Sarjana Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
luqmanarifin@students.itb.ac.id

***Abstrak* — Barisan rekursif adalah permasalahan yang sering dijumpai dalam bidang ilmu matematika diskrit. Contoh barisan rekursif yang paling popular adalah barisan bilangan fibonacci. Untuk menghitung bilangan ke-n dalam barisan bilangan fibonacci, salah satu solusi *straightforward* adalah melakukan iterasi secara satu-persatu untuk tiap 1 <= i <= n. Tentu saja kompleksitas algoritma ini adalah O(n). Nyatanya, barisan bilangan ini dapat diselesaikan dengan menggunakan pemangkatan matriks, yang kemudian dengan teknik *divide and conquer* dapat diselesaikan dengan kompleksitas logaritmik. Bagaimana mereduksi kompleksitas permasalahan ini?**

Kata kunci* — barisan rekursif, *divide and conquer*, kompleksitas, logaritmik, *matrix exponentiation

I. PENDAHULUAN

Permasalahan barisan bilangan rekursif adalah permasalahan umum di bidang matematika diskrit. Ada beberapa solusi dalam menentukan barisan bilangan rekursif. Salah satunya adalah solusi O(n) yaitu solusi secara iteratif menggunakan *dynamic programming*.

Kemudian masalah ini dapat direduksi menggunakan pemangkatan matriks sehingga kompleksitas dapat dikompresi menjadi O(log n). Cara ini memanfaatkan teknik *divide and conquer*.

Nilai ke-n dari barisan bilangan rekursif (misalnya barisan bilangan fibonacci) bisa sangat besar sekali. Oleh karena itu, penyelesaian barisan bilangan erat kaitannya dengan operasi *BigMod*. Kita juga mengerti bahwa struktur data dalam bahasa pemrograman, semisal *int* atau *long long* pada bahasa C memiliki batas tertentu. Struktur data int dapat menampung hanya hingga $2^{31} - 1$, sementara long long hingga $2^{63} - 1$.

Makalah ini tidak membahas angka eksak hasil barisan bilangan rekursif karena hasilnya bisa sangat besar dan tidak dapat ditampung dengan struktur data *built-in* pada bahasa pemrograman spesifik. Makalah ini hanya mengulas teknik penghitungannya saja. Teknik ini dapat menghasilkan hasil sisa pembagian dengan bilangan tertentu untuk barisan bilangan ke-n yang cukup besar.

II. DASAR TEORI

Berikut adalah dasar teori yang melandasi ditulisnya makalah ini.

A. *Barisan Bilangan Rekursif*

Barisan bilangan rekursif merupakan barisan bilangan yang memiliki relasi rekurens terhadap bilangan bersuku sebelumnya. Barisan bilangan rekursif juga memiliki basis.

Secara singkatnya, barisan bilangan rekursif adalah barisan bilangan yang memiliki relasi rekurens. Contoh paling familiar dari barisan bilangan rekursif adalah barisan bilangan fibonacci.

Relasi rekurens dari bilangan fibonacci adalah sebagai berikut.

$$F(n) = 0, \quad n = 1$$
$$F(n) = 1, \quad n = 2$$
$$F(n) = F(n-1) + F(n-2), \quad n > 2$$

Kemudian, semisal diketahui barisan bilangan berikut.

1, 3, 6, 10, 15, 21, 28, 35, 42, ...

Barisan bilangan di atas bukan merupakan barisan bilangan rekurens karena tidak memiliki relasi rekurens antar suku-sukunya yang berdekatan. Barisan bilangan tersebut dapat diketahui solusinya langsung yakni

$$U_n = \frac{n(n+1)}{2}$$

Contoh barisan bilangan rekursif yang lain adalah berikut. Diketahui

$$H(0) = 0$$
$$H(1) = 3$$
$$H(2) = 4$$
$$H(n) = 2H(n-3) + H(n-2) + H(n-1), \quad n > 2$$

Maka barisan bilangan rekursifnya adalah.

0, 3, 4, 7, 17, 32, 56, ...

Supaya bahasan tidak melebar, makalah ini dibatasi hanya dalam penyelesaian barisan bilangan rekursif, tidak untuk barisan bilangan lainnya.

B. Kompleksitas

Kompleksitas algoritma dibagi menjadi dua : kompleksitas ruang dan kompleksitas waktu.

Kompleksitas ruang, S(n), adalah memori yang digunakan oleh struktur data yang terdapat dalam algoritma yang ditunjukkan sebagai fungsi dari n yang mana n adalah masukan input.

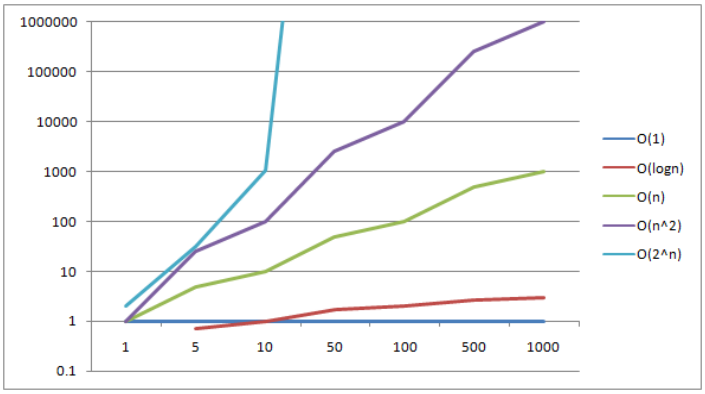
Kompleksitas waktu, T(n), adalah jumlah tahapan komputasi trivial yang dilakukan algoritma hingga proses selesai sebagai fungsi dari n. Sebagai contoh, algoritma brute force dalam pencarian nilai minimum dari n bilangan adalah T(n).

Sebagai batas atas, umum digunakan notasi big-O yang menunjukkan kompleksitas waktu algoritma dalam kasus terburuk (*worst case*). Contohnya dalam algoritma pencarian sebuah nilai dalam barisan bilangan tidak terurut dengan teknik *bruteforce*. T(n) nya tidak stabil, antara T(1) hingga T(n), tapi apabila dinyatakan dalam notasi big-O, algoritma ini memiliki kompleksitas O(n).

Berikut penulis lampirkan kompleksitas algoritma beserta istilah umumnya dan grafik perkembangan kompleksitas algoritma.

Kompleksitas	Istilah
O(1)	Constant complexity
O(log n)	Logarithmic complexity
O(n)	Linear complexity
O(n log n)	Linearithmic complexity
O(n^a), a >= 2	Polynomial complexity
O(b^n), b >= 2	Exponential complexity
O(n!)	Factorial complexity

Tabel 2. 1. Algoritma dan Istilah Umumnya



Grafik 2.2. Perkembangan kompleksitas algortima

C. Divide and Conquer

Divide and Conquer adalah teknik umum yang sering ditemui di dunia komputasi. Ide dasar dari teknik ini adalah membagi permasalahan menjadi beberapa permasalahan yang lebih kecil yang umumnya mirip, kemudian menyelesaikannya secara terpisah, lalu menggabungkan solusi hasil dari penyelesaian tersebut.

Divide : membagi permasalahan menjadi dua atau lebih permasalahan yang sama.

Conquer : selesaikan masalah-masalah yang telah dipecah secara sekursif. Umumnya masalah hasil pemecahan masih merupakan masalah yang sama.

D. Perkalian Matriks

Perkalian matriks dapat dilakukan secara iteratif dengan algoritma berkompleksitas O(n^3). Berikut adalah implementasi dalam bahasa C++.

```
1   #define SIZE 100
2
3   int a[SIZE][SIZE];
4   int b[SIZE][SIZE];
5   int ret[SIZE][SIZE];
6
7   void multiply() {
8       for(int i = 0; i < SIZE; i++)
9           for(int j = 0; j < SIZE; j++) {
10              ret[i][j] = 0;
11              for(int k = 0; k < SIZE; k++) {
12                  ret[i][j] += a[i][k] * b[k][j];
13              }
14          }
15  }
```

Gambar 2.3. Implementasi algoritma perkalian matriks

Algoritma perkalian matriks tidak hanya ini, namun juga bisa dilakukan dengan algoritma Strassen yang memenfaatkan divide and conquer dengan kompleksitas sekitar O($n^{2.8}$), namun makalah ini tidak menggunakan algoritma tersebut.

Perkalian matriks ini akan berguna dalam pemangkatan matriks, yang mana dalam proses pemangkatan akan banyak bersinggungan dengan perkalian antar dua matriks.

III. PENYELESAIAN BARISAN REKURSIF SECARA LINEAR

Andaikan kita diberi barisan bilangan rekursif dengan dengan basis dan rekurens sebagai berikut

$$H(1) = 1$$
$$H(2) = 3$$
$$H(3) = 5$$
$$H(n) = 2H(n-3) + 2H(n-2) + H(n-1)$$

kemudian diminta untuk mencari H(1000). Ada dua cara penyelesaian utama yakni bottom-up atau top-down.

Penyelesaian bottom-up, sesuai dengan namanya, kita akan membangun nilai fungsi H secara menaik dari i = 1 hingga i = n. Cara ini termasuk cara intuitif karena solusinya yang *straightforward*.

Berikut adalah implementasi bottom-up dalam bahasa

C++ untuk permasalahan di atas.

```
int find(int n) {
    int num[n + 1];
    num[1] = 1;
    num[2] = 3;
    num[3] = 5;
    for(int i = 4; i <= n; i++) {
        num[i] = 2*num[i - 3]
               + 2*num[i - 2]
               +   num[i - 1];
    }
    return num[n];
}
```

Gambar 3.1. Implementasi bottom-up O(n) dalam bahasa C++

Kompleksitas solusi di atas adalah

T(n) = max(n – 3, 1)
~ O(n)

Permasalahan ini juga bisa diselesaikan secara top-down, yakni penyelesaian secara rekursif. Untuk menghindari penghitungan fungsi yang sama sebanyak lebih dari sekali, kita bisa melakukan penyimpanan nilai / *cache*, setiap kali sebuah nilai sudah dihitung. Teknik ini juga sering disebut dengan *dynamic programming*.

Ide utama untuk menghindari penghitungan ganda dari top-down ini, ketika fungsi untuk sebuah paramater telah dihitung, simpan ke dalam array, dan tandai dengan bahwa parameter tersebut pernah dihitung. Ketika ada permintaan terhadap parameter yang sama, cukup keluarkan isi array tanpa harus menghitung kembali nilai kembaliannya.

```
#define SIZE 10000

bool visited[SIZE];
int cache[SIZE];

int find(int n) {
    switch(n) {
        case 1 : return 1; break;
        case 2 : return 3; break;
        case 3 : return 5; break;
    }
    if(visited[n]) return cache[n];
    visited[n] = true;
    cache[n] = 2*find(n - 3)
             + 2*find(n - 2)
             +   find(n - 1);
    return cache[n];
}

int main() {
    memset(visited, false, sizeof(visited));
    cout << find(1000) << endl;
    return 0;
}
```

Gambar 3.2. Implementasi top-down O(n) dalam bahasa C++

Kompleksitas solusi di atas adalah

$$T(1) = 1$$
$$T(2) = 1$$
$$T(3) = 1$$
$$T(n) = T(n-1) + 1$$

Relasi rekurens penghitungan kompleksitas hanya dihitung T(n) = T(n - 1) + 1 karena penghitungan **find(n)** untuk tiap nilai n hanya diproses sekali, akibatnya :

T(n) = n
~ O(n)

Berikut adalah data *running time* kedua program tersebut pada komputer dengan spesifikasi prosesor Intel Core i5 Asus A46C, compiler MinGW.

Input n	Bottom-Up	Top-Down
1000	**1 ms**	**0 ms**
10000	**1 ms**	**2 ms**
100000	**2 ms**	**3 ms**
1 juta	**11 ms**	***Stack overflow***
5 juta	**30 ms**	***Stack overflow***
10 juta	**54 ms**	***Stack overflow***
50 juta	**253 ms**	***Stack overflow***
100 juta	**501 ms**	***Stack overflow***
500 juta	**2449 ms**	***Stack overflow***
1 milyar	**4870 ms**	***Stack overflow***

Tabel 3.3. Komparasi *running time bottom-up* dan *top-down*

Kedua solusi di atas baik solusi bottom-up maupun top-down memiliki kompleksitas linear. Artinya untuk n yang berkembang secara linear, banyak tahapan operasi yang dilakukan algoritma juga berkembang secara linear. Algoritma ini tidak cukup mangkus dan sangkil untuk menyelesaikan n yang sangat besar. Bahkan solusi *top down* mendapat stack-overflow karena rekursif yang terlalu dalam.

Akibatnya, dibutuhkan optimasi lagi supaya pencarian barisan bilangan ke-n menjadi lebih cepat.

IV. Konversi Masalah Menjadi Pemangkatan Matriks

Telah disinggung di atas bahwa pencarian secara linear/sekuensial tidak cukup efisien untuk mencari barisan bilangan ke-n untuk n yang cukup besar.

Kenyataannya, permasalahan pencarian barisan bilangan ke-n ini bisa dikonversi menjadi pemangkatan matriks.

Andaikan ada barisan bilangan rekursif seperti berikut.

1, 3, 5, 13, 29, 65, 149, 337, 765, 1737, 3941, 8945, ...

dengan relasi rekurens dan basis di bawah.

$$H(1) = 1$$
$$H(2) = 3$$
$$H(3) = 5$$
$$H(n) = 2H(n-3) + 2H(n-2) + H(n-1)$$

Kemudian kita memiliki matriks berisi basis dari barisan bilangan rekursif tersebut.

$$A_1 = \begin{pmatrix} h_1 & h_2 & h_3 \end{pmatrix}$$

Ide utama dari konversi permasalahan ini adalah bagaimana caranya supaya matriks 1 x 3 ini dapat diubah menjadi matriks berisi elemen berikutnya.

$$A_2 = \begin{pmatrix} h_2 & h_3 & h_4 \end{pmatrix}$$

Dengan perkalian matriks, ternyata matriks A_1 bisa dikonversi menjadi matriks A_2. Tapi masalahnya, dikali dengan matriks apa?

Kita perlu melakukan observasi lebih lanjut. Perhatikan bahwa dua elemen pertama dalam matriks A_2 yaitu h_2 dan h_3, sebenarnya sudah terdapat dalam matriks A_1. Yang dapat kita lakukan adalah "mengambil" dua elemen berikut dari matriks A_1. Lalu bagaimana dengan h_4?

Jangan lupakan bahwa kita memiliki relasi rekurens

$$H(n) = 2H(n-3) + 2H(n-2) + H(n-1)$$

untuk n = 4, maka

$$h_4 = 2h_1 + 2h_2 + h_3$$

Catat bahwa elemen penyusun h_4 yakni h_1, h_2, dan h_3 adalah elemen-elemen milik matriks A_1. Dari sini kita dapat simpulkan bahwa kita bisa membangun matriks A_2 dengan merekayasa matriks sedemikian rupa sehingga perkalian antara matriks A_1 dan *matriks rekayasa* menghasilkan matriks A_2.

Dan ternyata

$$= \begin{pmatrix} h_1 & h_2 & h_3 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 2 \\ 1 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix}$$
$$= \begin{pmatrix} h_2 & h_3 & 2h_1 + 2h_2 + h_3 \end{pmatrix}$$
$$= \begin{pmatrix} h_2 & h_3 & h_4 \end{pmatrix}$$
$$= A_2$$

Dari sini didapat matriks rekayasa, kita sebut nantinya dengan **matriks transisi**.

$$T = \begin{pmatrix} 0 & 0 & 2 \\ 1 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix}$$

Matriks transisi memiliki properti-properti sebagai berikut.

Kolom 1 : mengendalikan transisi untuk elemen ke-1
Kolom 2 : mengendalikan transisi untuk elemen ke-2
Kolom 3 : mengendalikan transisi untuk elemen ke-3
Atau secara umum dapat dikatakan untuk kolom 1 <= i <= Ukuran matriks
Kolom i : mengendalikan transisi untuk elemen ke-i.

Properti lain dalam matriks transisi adalah :

1. Kolom terakhir selalu berisi relasi rekurens.
2. Ukuran matriks selalu persegi, artinya jumlah kolom sama dengan jumlah baris.
3. Ukuran matriks bergantung pada selisih terbesar pada elemen yang terlibat dalam relasi rekurens. Misalnya dalam kasus di atas, elemen ke-n dan elemen ke-(n - 3) terlibat dalam relasi rekurens, akibatnya ukuran matriks adalah selisihnya yakni 3 x 3.

Dengan cara serupa dapat dicari matriks A_3 dan A_4 karena

$$A_2 \cdot T = \begin{pmatrix} h_2 & h_3 & h_4 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 2 \\ 1 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix}$$
$$= \begin{pmatrix} h_3 & h_4 & 2h_2 + 2h_3 + h_4 \end{pmatrix}$$
$$= \begin{pmatrix} h_3 & h_4 & h_5 \end{pmatrix}$$
$$= A_3$$

$$A_3 \cdot T = \begin{pmatrix} h_3 & h_4 & h_5 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 2 \\ 1 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix}$$
$$= \begin{pmatrix} h_4 & h_5 & 2h_3 + 2h_4 + h_5 \end{pmatrix}$$
$$= \begin{pmatrix} h_4 & h_5 & h_6 \end{pmatrix}$$
$$= A_4$$

Secara umum transisi ini dapat diekspresikan menjadi

$$A_n \cdot T = \begin{pmatrix} h_n & h_{n+1} & h_{n+2} \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 2 \\ 1 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix}$$
$$= \begin{pmatrix} h_{n+1} & h_{n+2} & 2h_n + 2h_{n+1} + h_{n+2} \end{pmatrix}$$
$$= \begin{pmatrix} h_{n+1} & h_{n+2} & h_{n+3} \end{pmatrix}$$
$$A_n \cdot T = A_{n+1}$$

Untuk semua n >= 1, dapat dicari matriks A_n-nya. Berikut adalah bukti menggunakan induksi matematika.

Basis : Elemen pada matriks A_1 adalah elemen basis pada barisan bilangan rekursif.

Rekurens : Andai A_n adalah matriks dengan elemen barisan bilangan rekursif. Perkalian A_n dengan matriks transisi, menghasilkan A_{n+1} yang juga merupakan matriks dengan elemen barisan bilangan rekursif. Terbukti bahwa dapat dicari matriks A_n untuk semua nilai n.

$$A_{n+1} = A_n \cdot T$$

Dari persamaan di atas, kita dapat menurunkan persamaan berikut.

$$A_2 = A_1 \cdot T$$

$$A_3 = A_2 \cdot T = A_1 \cdot T^2$$

$$A_4 = A_3 \cdot T = A_1 \cdot T^3$$

$$A_5 = A_4 \cdot T = A_1 \cdot T^4$$

$$\cdots$$

$$A_n = A_{n-1} \cdot T = A_1 \cdot T^{n-1}$$

> Note : Elemen pertama pada matriks **A_n** adalah barisan bilangan rekursif ke-n **(h_n)**

Dengan ini untuk mencari barisan bilangan ke-n (h_n), cukup bentuk matriks basis A_1 dan matriks transisi T kemudian cari elemen pertama dari matriks $A_1.T^{n-1}$.

Perkalian matriks bersifat asosiatif, artinya untuk operasi perkalian yang beruntutan, kita bisa melakukan yang mana saja terlebih dahulu karena hasilnya sama. Begitu juga dalam kasus komputasi untuk persamaan di atas. Karena perkalian matriks bersifat asosiatif, T^{n-1} bisa dihitung terlebih dahulu kemudian baru dikalikan dengan A_1. Matriks hasil (A_n) yang akan diperoleh sama saja.

V. Implementasi dan Analisis Kompleksitas

Sesuai yang telah diulas oleh M. Dikra Prasetya dalam makalah Matematika Diskritnya pada [2], operasi pemangkatan dapat dilakukan secara linear yakni secara iteratif dengan kompleksitas O(n), atau memanfaatkan teknik *divide and conquer* dengan kompleksitas O(log n).

Penggunaan algoritma pemangkatan secara iteratif tidak akan *make sense* karena kompleksitas yang dihasilkan sama dengan solusi linear seperti yang telah kita singgung pada bab 3 di atas. Untuk mendapat kompleksitas yang mangkus dan sangkil, kita perlu menggunakan teknik pemangkatan dengan *divide and conquer*.

Dengan teknik *divide and conquer*. Operasi pemangkatan matriks seluruhnya akan membutuhkan cost sebesar O(log n) dengan n adalah berapa pangkatnya. Namun, setiap operasi perkalian matriks dilakukan, dibutuhkan kompleksitas $O(s^3)$ dengan s adalah ukuran matriks yang akan dikalikan (perkalian matriks telah kita singgung sebelumnya di bab 2 Dasar Teori).

Akibatnya kompleksitas total yang dibutuhkan algoritma pencarian barisan bilangan rekursif menggunakan pemangkatan matriks adalah

$$T(n) = s^3 \cdot \log(n)$$

$$\sim O(s^3 \cdot \log(n))$$

Berikut adalah implementasi algoritma pencarian barisan bilangan rekursif dengan pemangkatan matriks dalam bahasa C++.

```
27  #define SIZE 3
28
29  int t[SIZE][SIZE];
30  int a[SIZE];
31  int temp[SIZE][SIZE];
32
33  void multiply_temp_temp() {
34      int ret[SIZE][SIZE];
35      for(int i = 0; i < SIZE; i++)
36          for(int j = 0; j < SIZE; j++) {
37              ret[i][j] = 0;
38              for(int k = 0; k < SIZE; k++) {
39                  ret[i][j] += temp[i][k] * temp[k][j];
40              }
41          }
42      for(int i = 0; i < SIZE; i++)
43          for(int j = 0; j < SIZE; j++) {
44              temp[i][j] = ret[i][j];
45          }
46  }
47
48  void multiply_t_with_temp() {
49      int ret[SIZE][SIZE];
50      for(int i = 0; i < SIZE; i++)
51          for(int j = 0; j < SIZE; j++) {
52              ret[i][j] = 0;
53              for(int k = 0; k < SIZE; k++) {
54                  ret[i][j] += t[i][k] * temp[k][j];
55              }
56          }
57      for(int i = 0; i < SIZE; i++)
58          for(int j = 0; j < SIZE; j++) {
59              temp[i][j] = ret[i][j];
60          }
61  }
```

Gambar 5.1. Implementasi solusi logaritmik menggunakan pemangkatan matriks (1)

```
62
63  void multiply_a_with_t_power() {
64      int ret[SIZE];
65      for(int j = 0; j < SIZE; j++) {
66          ret[j] = 0;
67          for(int k = 0; k < SIZE; k++) {
68              ret[j] += a[k] * temp[k][j];
69          }
70      }
71      for(int j = 0; j < SIZE; j++) {
72          a[j] = ret[j];
73      }
74  }
75
76  void power_t(LL n) {
77      if(n == 0) return;
78      power_t(n/2);
79      multiply_temp_temp();
80      if(n % 2) {
81          multiply_t_with_temp();
82      }
83  }
84
```

Gambar 5.2. Implementasi solusi logaritmik menggunakan pemangkatan matriks (2)

```
84
85  int main(void)
86  {
87      t[0][0] = 0; t[0][1] = 0; t[0][2] = 2;
88      t[1][0] = 1; t[1][1] = 0; t[1][2] = 2;
89      t[2][0] = 0; t[2][1] = 1; t[2][2] = 1;
90
91      a[0] = 1; a[1] = 3; a[2] = 5;
92
93      temp[0][0] = 1; temp[0][1] = 0; temp[0][2] = 0;
94      temp[1][0] = 0; temp[1][1] = 1; temp[1][2] = 0;
95      temp[2][0] = 0; temp[2][1] = 0; temp[2][2] = 1;
96
97      LL n = 1e18;
98      power_t(n - 1);
99      multiply_a_with_t_power();
100     cout << a[0] << endl;
101     return 0;
102 }
103
```

Gambar 5.3. Implementasi solusi logaritmik menggunakan pemangkatan matriks (3)

Berikut adalah data *running time* program tersebut pada komputer dengan spesifikasi prosesor Intel Core i5 Asus A46C, compiler MinGW.

Input n	**Running time**
1000	**0 ms**
1 juta	**1 ms**
1 milyar	**1 ms**
1 triliun	**1 ms**
1000 triliun	**1 ms**
1 juta triliun	**2 ms**

Tabel 5.4. Perkembangan *running time* solusi logaritmik menggunakan pemangkatan matriks

Algoritma ini berjalan sangat cepat karena perkembangan kompleksitasnya logaritmik. Bisa dikatakan algoritma ini cukup mangkus dan sangkil untuk berjalan pada n *gigantic*.

VI. KESIMPULAN

Pemangkatan matriks dengan teknik *divide and conquer* dapat digunakan untuk mereduksi kompleksitas dalam permasalahan pencarian barisan bilangan rekursif. Teknik ini berjalan pada kompleksitas $O(s^3 \log n)$ sehingga sangat efisien untuk mengkomputasi pada nilai n yang besar.

REFERENSI

[1] K. H. Rosen. *Discrete Mathematics and Its Applications 7th*. New York: McGraw-Hill, 2012.

[2] M.D. Prasetya. "Analisis Kompleksitas Kalkulasi Operasi Pangkat" (Makalah Matematika Diskrit). Bandung: Program Studi Teknik Informatika Sekolah Teknik Elektro dan Informatika Institut Teknologi Bandung, 2012.

[3] H. Schildt. *C++ The Complete Reference 2nd Edition*. California: McGraw-Hill, 1995.

[4] E. Darmawan. *Pemrograman Dasar C - Java – C#*. Bandung: Informatika, 2009.

[5] T. H. Cormen. *Introduction to Algorithm 3rd Edition*. Massachusetts: The MIT Press, 2009.

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 2014

Luqman Arifin Siswanto
13513024