Analisis Kompleksitas Algoritma dalam Operasi BigMod

Calvin sadewa / 13512066
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
calvin.sadewa@students.itb.ac.id

***Abstract*—Operasi modulo adalah salah satu operasi yang sering digunakan dalam komputer, untuk itu, dibutuhkan suatu algoritma yang dapat menyelesaikan operasi modulo dengan cepat. Makalah ini membahas tentang algoritma operasi modulo dalam bentuk a^b mod c, dan membandingkan kompleksitas waktunya.**

***Index Terms*—modulo, kompleksitas waktu, BigMod, exponen**

I. PENDAHULUAN

Operasi modulo adalah salah satu operasi yang bisa dibilang selalu ada pada komputer, selain itu, operasi ini juga memiliki banyak kegunaan di bidang kriptografi , salah satu contohnya ada di algoritma RSA. Algoritma hashing juga bisa dibuat dengan operasi modulo.

Operasi BigMod merupakan operasi modulo dalam bentuk $a^b mod\ c$, masalah yang paling sering muncul dalam operasi ini adalah seringkali nilai a^b sangat besar sehingga melebihi nilai yang bisa ditampung di dalam variable, akibatnya , Operasi BigMod harus dipecah dulu menjadi Operasi-operasi yang lebih kecil.

Dalam makalah ini, kita akan membahas berbagai algoritma yang dapat digunakan dalam menghitung operasi BigMod, ukuran dari variable tidak akan diperhitungkan, dan kompleksitas waktu dinilai dari berapa banyak operasi modulo yang dilakukan.

II. TEORI TERKAIT

A. Operasi Modulo

Operasi modulo dapat di definisikan sebagai :

$a = b\ mod\ c$ jika $b = k.c + a$, dengan a,b,c,k merupakan bilangan bulat dan c > a >= 0

Perhatikan bahwa konsep diatas biasanya tidak berlaku di komputer, karena komputer memiliki konvensi tersendiri dalam operasi modulo. Untuk lebih jelas silahkan lihat deskripsi Knuth[2] dan bandingkan dengan Raymond T.Boute[3]

Selain itu, ada juga konsep kongruen, dimana :

$b \equiv a(mod\ c)$ jika dan hanya jika $a\ mod\ c = b\ mod\ c$

Perhatikan symbol '≡' menyatakan kongruen

Ada juga arithmatika modular, sebagian akan di tuliskan di sini

1. Jika $a \equiv b\ (mod\ m)$ dan $c \equiv d\ (mod\ m)$ maka
   a. $(a + c) \equiv (b + d)(mod\ m)$
   b. $a.c \equiv b.d\ (mod\ m)$
2. Jika $a \equiv b\ (mod\ m)$ dan c adalah bilangan bulat, maka
   a. $(a + c) \equiv (b + c)(mod\ m)$
   b. $a.c \equiv b.c\ (mod\ m)$
   c. $a^k \equiv b^k(mod\ m)$, k bilangan bulat >= 0

B. Kompleksitas

Kompleksitas dapat didefinisikan sebagai berapa banyak kebutuhan sumber daya (resource) untuk menyelesaikan suatu masalah yang tergantung dari berapa banyak jumlah inputan.

Sumber daya secara umum dapat dibagi menjadi dua, yaitu waktu (time) dan ruang (space), seperti yang di tuliskan di abstrak, dalam makalah ini kompleksitas waktu lebih sering dibicarakan dibandingkan kompleksitas ruang

Kompleksitas waktu, T(n), diukur dari berapa banyak operasi khas yang terjadi didalam menyelesaikan masalah, dan n merepresentasikan banyaknya inputan. Terkadang seseorang lebih suka melihat pengaruh T(n) pada n besar dan kurang mempedulikan sisanya, dari sini muncul definisi big-o dimana

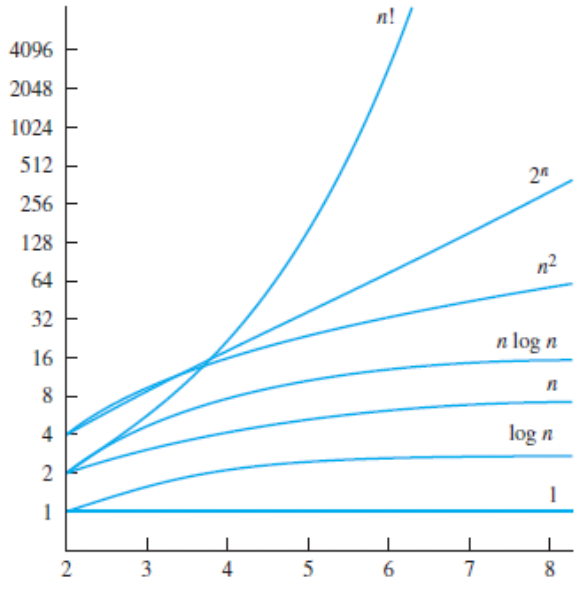
$$T(n) = O(f(n))$$

Jika $T(n) \leq C.f(n)$ untuk

C bilangan positif sembarang dan $n > n_0$, n_0 sembarang

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Tabel 2.1 Kompleksitas Algoritma dan Istilahnya[1]



Gambar 2.1 Pertumbuhan Komplesitas terhadap input[1]

C. Operasi Pangkat

Karena pangkat merupakan kharakteristik khusus dalam operasi BigMod , maka akan di jelaskan sedikit tentang identitas – identitas dari operasi pemangkatan.

$$a^b . a^c = a^{b+c}$$

$$\frac{a^b}{a^c} = a^{b-c}$$

$$(a^b)^c = a^{b.c}$$

$$(a.b)^c = a^c . b^c$$

$$a^{-b} = \frac{1}{a^b}$$

III. ANALISIS ALGORITMA

A. Naif

Pemecahan masalah dengan algoritma ini pada dasarnya menggunakan sifat modular yang ke 2.b, dengan sifat itu, kita bisa menuliskan

$$a^b \equiv a.a^{b-1} \ (mod\ c), \text{b} > 0$$

Dimana a diproses secara terpisah dari a^{b-1} , Algoritmanya adalah sebagai berikut :

1. $a^b \equiv (a \ mod \ c).a^{b-1} \ (mod\ c)$, b > 0
2. $a^b \equiv k.a^{b-1} \ (mod\ c)$, b > 0, k = $a \ mod \ c$
3. $a^b \equiv (k.a).a^{b-2} \ (mod\ c)$
4. Kembali ke langkah 1 kalau b > 0, berhenti jika tidak .

[1] Gambar di ambil dari Discreet Mathematics and Its Application ed.7th hal.211

Pada realisasinya, algoritma ini bisa dibuat dalam 2 buah bentuk, yaitu bentuk linier dan bentuk rekursif.

Inilah algoritma linier yang di implementasikan dengan bahasa pemograman C

```
1   int BigModNaifLinier(int a,int b,int c)
2   {
3       int e;
4       long long Hasil = 1;
5       for (e=b;e>0;e--)
6       {
7           Hasil = (Hasil * a) % c;
8       }
9       return (int) Hasil;
10  }
```

Gambar 3.1 Cuplikan kode BigMod implementasi naïf Linier

Dalam kode di atas, ada variable *e* sebagai ganti *b* dan variable *hasil* untuk menyimpan keluaran. Pada baris ke 5, ada kalang for, dimana kalang tersebut akan mengulang sebanyak b-1 kali, didalam kalang, ada operasi * dan operasi %, karena kita hanya melihat operasi % sebagai operasi yang khas, maka setiap kali pengulangan dilakukan satu kali operasi khas, jadinya

$$T(n) = n - 1, b = n \text{ dengan O(n)}$$

Persamaan diatas sama untuk worst-case dan best-case

Setelah melihat solusi linier diatas, kita bisa membuat solusi rekursif dengan fungsi BigMod :

$$BigMod(a,b,c)\begin{cases} 1, b = 0 \\ ((a * BigMod(a,b-1,c))mod\ c, b \neq 0 \end{cases}$$

Berikut implementasinya dalam C

```
1   int BigModNaifRekursif (int a,int b,int c)
2   {
3       long long Hasil;
4       if (b==0) {return 1;}
5       else
6       {
7           Hasil = (a*BigModNaifRekursif(a,b-1,c)) % c;
8           return Hasil;
9       }
10  }
```

Gambar 3.2 Cuplikan kode pendekatan Naif rekurens

Perhatikan bagian rekurensnya, ada operasi % di dalamnya, ini menandakan bahwa setiap kali rekurens dilakukan satu kali operasi khas, kalau diperhatikan basis nya (b=0) dan setiap rekurensnya mengurangi b dengan satu, maka dapat dituliskan

$$T(n) = n - 1, b = n \text{ dengan O(n)}$$

Tidak ada perbedaan komplesitas dari sisi linier maupun sisi rekurens, namun dari segi kompleksitas ruang pendekatan secara linier lebih bagus karena hanya menggunakan 2 variabel tambahan, sedangkan rekurens

membuat variable “a,b,c” baru setiap kali rekurens dipanggil.

B. Mengexponensikan dengan kuadrat

Pendekatan pada cara ini merupakan paradigm divide and conquer, dimana permasalahan di pecah kedalam masalah yang lebih kecil. Sebagai permulaan, kita bisa menuliskan :

$$a^b \equiv \begin{cases} a^{b/2}.a^{b/2}\ (mod\ c), b\ bilangan\ genap > 0 \\ a.a^{(b-1)/2}.a^{(b-1)/2}\ (mod\ c), b\ ganjil > 1 \end{cases}'$$

Di rumus diatas kita bisa melihat bahwa permasalahan diubah kedalam bentuk yang lebih kecil, kemudian kita bisa memecahkan masalah tersebut seperti pada algoritma naïf.

Sama seperti pada pendekatan naïf, pendekatan ini bisa diimplementasikan dengan 2 cara, yaitu linier dan rekursif, mari kita lihat implementasi linier dalam C

```
1  int BigModSquareLinier(int a, int b, int c)
2  {
3      long long a2 = a;
4      int e;
5      long long Hasil = 1;
6      for (e=b;e>0;e=e/2)
7      {
8          if (e & 1)
9          {
10             e = e - 1;
11             Hasil = (Hasil * a2) %c;
12         }
13         a2 = (a2*a2) % c;
14     }
15     return (int) Hasil;
16 }
```

Gambar 3.3 kode kuadrat dalam linier

Dalam solusi ini, digunakan identitas.

$$a^b = (a^2)^{b/2}, b\ genap > 0$$
$$a^b \equiv (k)^{\frac{b}{2}}\ (mod\ c), b\ genap > 0, k = a^2 mod\ c$$

Perhatikan bahwa dalam implementasi , variable a2 merepresentasikan k, dan e merepresentasikan nilai b yang tereduksi, e & 1 menyatakan apakah variable e ganjil atau tidak, dimana jika ganjil nilai e akan dikurang dengan satu dan hasil akan dikalikan dengan a2.

Kita dapat menghitung best-case dengan mengasumsikan nilai b merupakan bentuk dari 2^i, dengan i bilangan bulat positif, dengan begitu, maka algoritma tidak akan pernah memasuki kondisi if pada baris ke 8 kecuali pada i=0, sehingga pada kalang for di baris ke 6, hanya akan dilakukan satu kali operasi modulo kecuali pada e=1, yang akan dilakukan 2 kali, dan banyak pengulangan adalah $\log_2 b + 1$, oleh karena itu, dapat dituliskan

$$T_{best}(n) = 1 + \log_2(n+1), n = b$$
$$T_{best}(n) = O_{best}(\log n)$$

Untuk worst-case , kompleksitas dapat dicari dengan mengandaikan bahwa untuk setiap kali pengulangan, kondisi if di baris ke 8 selalu tercapai, akibatnya setiap kali pengulangan ada 2 operasi modulo , kompleksitas dapat dituliskan sebagai :

$$T_{worst}(n) = 2\log_2(n+1), n = b$$
$$T_{worst}(n) = O_{worst}(\log n)$$

Untuk average, kita dapat mengasumsikan bahwa kondisi if terjadi setengah kali pengulangan , sehingga kompleksitas adalah :

$$T_{average}(n) = (1\tfrac{1}{2})log_2(n+1), n = b$$
$$T_{average}(n) = O_{average}(\log n)$$

Setelah melihat implementasi linier, mari kita lihat bentuk rekursif, bentuk ini dibuat dari fungsi rekursif :

$$BigMod(a,b,c) = \begin{cases} 1, b = 0 \\ \left(BigMod\left(a,\frac{b}{2},c\right)^2\right) mod\ c \\ \left(a * BigMod\left(a,\frac{b}{2},c\right)^2\right) mod\ c \end{cases}$$

Kondisi ke 2 untuk b genap, kondisi ke 3 untuk b ganjil, pembagian disini dibuat dengan pembulatan kebawah

Implementasinya dalam C dapat dilihat di bawah

```
1  int BigModSquareRekursif(int a,int b,int c)
2  {
3      long long Temp;
4      if (b==0) {return 1;}
5      else
6      {
7          Temp = BigModSquareRekursif(a,b>>1,c);
8          Temp = (Temp * Temp)%c;
9          if (b&1) {return (a*Temp)%c;}
10         else {return Temp;}
11     }
12 }
```

Gambar 3.4 kode kuadrat dalam rekursif

Dalam baris ke 7 ada operasi “>>1”, dimana operasi itu ekuivalen dengan operasi “/2”, Variabel Temp digunakan untuk menghindari perhitungan rekurens yang berlebihan.

Dalam rekurens, selalu terdapat minimal satu operasi modulo dan maksimal dua jika kondisi if di baris ke 9 terpenuhi, selain itu, karena basis adalah 0, dan setiap kali terjadi rekuren b selalu dereduksi dengan b>>1, maka akan terjadi rekurens sebanyak banyak bit dari b, yang setara dengan $1 + \log_2 b + 1$, namun karena basis tidak memiliki operasi modulo, maka pengulangan yang memiliki makna hanya $\log_2 b + 1$.

Untuk mencari waktu tercepat, haruslah dipilih nilai b sedemikian rupa sehingga kondisi if di baris ke 9 minimal, yaitu dalam bentuk 2^i, sehingga setiap pengulangan hanya butuh satu kali operasi modulo kecuali saat i=0, dari sini, kompleksitas waktunya dapat dituliskan sebagai :

$$\mathrm{T}_{best}(n) = 1 + \log_2(n+1), n = b$$
$$T_{best}(n) = O_{best}(\log n)$$

Untuk mencari kompleksitas maksimum, kita perlu membuat nilai b sehingga setiap kali rekurens kondisi if terpenuhi, dengan ini kita bisa menuliskan :

$$T_{worst}(n) = 2\log_2(n+1), n = b$$
$$T_{worst}(n) = O_{worst}(\log n)$$

Pada kompleksitas rata-rata, kita memakai asumsi sama pada yang ada di linier, yaitu setiap kali rekurens ada peluang 50% kondisi if tidak terpenuhi, dari situ kita bisa mendapatkan :

$$T_{average}(n) = (1\tfrac{1}{2})log_2(n+1), n = b$$
$$T_{average}(n) = O_{average}(\log n)$$

Setelah di lakukan perbandingan, ternyata kompleksitas waktu untuk solusi linier dan solusi rekursif sama, oleh karena itu solusi linier lebih baik karena tidak memakan ruang terlalu banyak sedangkan solusi rekursif membutuhkan 3 ruang setiap kali rekurens dipanggil.

C. Pendekatan Left to Right

Pendekatan ini dimulai dengan identitas

$$b = 2^k + i, b > 0$$

Dimana k dan i merupakan bilangan bulat positif serta memenuhi persamaan b/2 > i >= 0.

Dengan memasukan identitas diatas ini kedalam persamaan BigMod, didapatkan :

$$a^b \equiv a^{2^k}.a^i \ (mod\ c)$$

Representasi biner pada komputer sangat cocok dengan deskripsi ini, oleh karena itu, algoritma ini cukup sering dipakai didalam komputer, pertama-tama, mari kita definisikan kembali b dalam notasi biner

$$b = \{b_{n-1}, b_{n-2}, \ldots., b_0\}_2$$

Kemudian algoritmanya adalah :

1. Hasil = 1;
2. i = n-1;
3. while (i ≥ 0) {
4. Hasil=Hasil^2 mod c;
5. If (b_i = 1) {
6. Hasil = (Hasil x a) mod c;
7. }
8. i = i - 1;
9. }

Dalam kalang while, terjadi pengulangan sebanyak jumlah bit dari dari b, yang setara dengan $\log_2 b + 1$, kemudian pada setiap pengulangan membutuhkan minimal 1operasi modulus dan maksimal 2 operasi modulus.

Untuk mencari waktu tercepat, haruslah bit pertama 1 dan sisanya 0, sehingga kondisi if cuma terjadi sekali,dapat di tuliskan

$$\mathrm{T}_{best}(n) = 1 + \log_2(n+1), n = b$$
$$T_{best}(n) = O_{best}(\log n)$$

Untuk mecari waktu terlambat, haruslah bitnya semua 1, sehingga kondisi if selalu terpenuhi ,kompleksitas dapat di tuliskan sebagai

$$T_{worst}(n) = 2\log_2(n+1), n = b$$
$$T_{worst}(n) = O_{worst}(\log n)$$

Untuk mencari waktu rata-rata di asumsikan distribusi 1 dan 0 dalam bit sama, sehingga kompleksitasnya adalah

$$T_{average}(n) = (1\tfrac{1}{2})log_2(n+1), n = b$$
$$T_{average}(n) = O_{average}(\log n)$$

Dalam ilmu komputer, pendekatan dengan kuadrat sering disebut sebagai pendekatan RL (Right to Left), karena ia melakukan perubahan dimulai dari bit terkanan. Namun, seperti yang telah dituliskan, dari segi kompleksitas waktu (ditinjau dari operasi modulus) , Pendekatan dan segi RL maupun LR menghasilkan kompleksitas waktu yang sama saja.

IV. KESIMPULAN

Dari algoritma-algoritma yang telah di coba, algoritma dengan pendekatan kuadrat dan algoritma dengan pendekatan LR merupakan algoritma terbaik dari segi kompleksitas waktu (dengan operasi modulo sebagai operasi khasnya), dari dua implementasi (linier dan rekursif) untuk algoritma naïf dan algoritma kuadrat, dua implementasi tersebut menghasilkan kompleksitas waktu yang sama saja.

DAFTAR PUSTAKA

[1] Rosen,Kenneth H. “Discreet Mathematics and Its Application” ed.7th ,McGraw-Hill

[2] Knuth, Donald. E. (1972). The Art of Komputer Programming. Addison-Wesley.

[3] Boute, Raymond T. (April 1992). "The Euclidean definition of the functions div and mod". ACM Transactions on Programming Languages and Systems (TOPLAS) (ACM Press (New York, NY, USA)) 14 (2): 127–144.

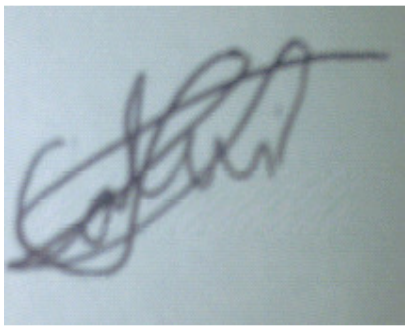
[4] http://eli.thegreenplace.net/2009/03/28/efficient-modular-exponentiation-algorithms/ 6.03PM 17/12/2013

[5] Guan, D.J. 2007 Efficient Algorithms for Computing Modular Exponentiation (Online), (guan.cse.nsysu.edu.tw/note/expn.pdf) diakses 15 Desember 2013).

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, 17 Desember 2013



Calvin sadewa / 13512066