Sieve of Eratosthenes dan Aplikasinya Dalam *Problem Solving*

Christianto - NIM : 1350003[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13510003@std.stei.itb.ac.id, handojo_christianto@yahoo.com

***Abstract*—Makalah ini akan membahas algoritma yang biasa digunakan untuk mendapatkan semua bilangan prima antara 1 sampai dengan batas tertentu, yaitu Sieve of Eratosthenes beserta contoh-contoh persoalan yang dapat diselesaikan dengan algoritma Sieve of Eratosthenes yang diadaptasi sedikit. Algoritma Sieve of Eratosthenes yang akan diberikan disini adalah versi normal dan versi yang sudah dioptimisasi. Semua algoritma yang ditulis pada makalah ini akan dianalisa komplesitas waktu dan memorinya.**

***Index Terms*—Sieve of Eratosthenes, bilangan prima, kompleksitas.**

1. Pendahuluan

Dalam beberapa permasalahan, diperlukan cara yang cepat untuk mencari beberapa bilangan prima yang terletak dalam rentang batas tertentu. Permasalahan mencari bilangan prima tersebut dapat digeneralisasi menjadi mencari semua bilangan prima antara 1 sampai dengan batas atas dari rentang tersebut, sebut saja N.

Cara *trial division*, yaitu mengetes semua bilangan tersebut untuk mencari semua prima tidaklah mangkus, karena cara terbaik untuk mengetes apakah sebuah bilangan prima atau komposite masih memerlukan pengecekan dengan semua bilangan dari satu sampai dengan akar dari bilangan tersebut, sehingga kompleksitas dari cara *tersebut* adalah O(N * $\sqrt{N}$), yang memakan waktu lama bahkan untuk ukuran N sebesar 1.000.000.

Dalam dunia komputer sudah ditemukan beberapa cara untuk mencari bilangan prima tersebut, cara yang termudah dinamakan *Sieve of Eratosthenes*, sesuai dengan nama orang yang menemukannya, yaitu Eratosthenes, ahli matematika dari Kirene yang hidup pada 276 SM-195 SM[1].

Sieve of Eratosthenes sendiri dapat dikembangkan dengan melihat properti-properti dari bilangan bulat dan prima, sehingga jumlah proses yang dilakukan dapat dikurangi.Dalam bab 3 akan dibahas dua buah optimisasi yang mudah untuk dilakukan, tetapi sudah memberikan perbaikan yang cukup signifikan. Masih ada optimisasi lainnya yang akan memberikan perbaikan tambahan untuk *Sieve of Eratosthenes.*

2. Algoritma untuk *Sieve of Eratosthenes*

Pada dasarnya, algoritma untuk *Sieve of Eratosthenes* adalah sebagai berikut :

1. Buat daftar bilangan dari 2 sampai dengan N.
2. Ambil bilangan terkecil dalam daftar tersebut, sebut saja X. X adalah bilangan prima yang kita cari, keluarkan dari daftar dan masukkan ke tempat lain.
3. Buang semua kelipatan X dari daftar tersebut.
4. Ulangi langkah 2 dan 3 sampai tidak ada bilangan yang tersisa dalam daftar.

Simulasi dari algoritma ini dapat dilihat pada [2]. Pada contoh tersebut, N adalah 120.

Dalam melakukan langkah-langkah di atas, perlu diperhatikan bahwa cara yang dipilih akan mempengaruhi kemangkusan dari algoritma ini. Bila cara yang dipilih adalah melakukan semua langkah tersebut secara harfiah, maka kompleksitas yang didapatkan akan menjadi sangat buruk, karena menghapus elemen daftar yang diimplementasikan dengan sebuah list linier akan membuang waktu sangat banyak.

Oleh karena itu, berikut ini adalah cara yang dipilih untuk menjalankan algoritma tersebut :

1. Buat sebuah larik A bertipe boolean, indeksnya antara 2 sampai dengan N. Isi semuanya dengan true.
2. Inisialisasi variabel bertipe integer I dengan 2.
3. Bila A_I bernilai true, maka untuk J = semua kelipatan I mulai dari 2*I sampai I * $\lfloor N/I \rfloor$, A_J dimasukkan nilai false. Bila A_I bernilai false jangan lakukan apa-apa.
4. Bila nilai I masih kurang dari N, maka I ditambah satu, kemudian ulangi dari langkah 3. Bila tidak berarti semua proses sudah selesai.

Pada akhir dari proses-proses yang dijelaskan di atas, semua indeks I dimana A_I bernilai true merupakan bilangan prima. Berikut ini adalah *pseudocode* untuk algoritma di atas :

```
Sieve-of-Eratosthenes(N)

for I ← 2 to N
    do A_I ← true

for I ← 2 to N
    do if (A_I = true)
        then for J ← 2 to floor(N / I)
            do A_I*J ← false
→ A
```

Contoh eksekusi algoritma di atas dapat ditemukan pada [2].

Untuk menghitung kompleksitas algoritma di atas, perhatikan bahwa bagian yang paling memakan waktu adalah loop bersarang pada bagian kedua. Pada bagian tersebut, jika I adalah bilangan prima, maka J akan dieksekusi sebanyak O($\lfloor N/I \rfloor$). Maka kompleksitas algoritma tersebut adalah (p adalah bilangan prima apapun):

$$\sum_{p \le N} \lfloor N/p \rfloor \le \sum_{p \le N} N/p$$
$$= N * \sum_{p \le N} 1/p$$
$$\approx N * \ln \ln N$$

Untuk penjelasan proses dari persamaan ketiga menjadi persamaan terakhir dapat dilihat di [3]. Dari persamaan di atas, terlihat bahwa kompleksitas waktu untuk algoritma ini adalah O(N ln ln N). Sedangkan kompleksitas memori dari algoritma tersebut adalah O(N), karena hanya diperlukan sebuah larik dengan indeks 2 sampai dengan N.

Dari penjelasan di atas, dapat disimpulkan bahwa *Sieve of Eratosthenes* dapat dijalankan dengan algoritma yang memiliki kompleksitas waktu O(N ln ln N) dan kompleksitas memori O(N).

3. Optimisasi Algoritma *Sieve of Eratosthenes*

Algoritma yang diberikan di atas sebetulnya masih dapat dikembangkan lebih lanjut untuk mendapatkan algoritma yang walaupun memiliki kompleksitas waktu yang kurang lebih sama, memiliki faktor konstan yang lebih baik. Pengembangan yang dilakukan didasarkan pada sebuah teorema dan sebuah lemma.

Teorema 3.1
Untuk setiap bilangan komposit c, akan terdapat minimal sebuah faktor prima dari c yang lebih kecil atau sama dengan $\sqrt{c}$.

Bukti - Karena c bilangan komposit, maka faktorisasi prima c terdiri dari minimal 2 bilangan prima. Ambil a dan b sebagai 2 hasil faktorisasi prima c yang berbeda. Salah satu dari a atau b pasti $\le \sqrt{c}$, karena bila tidak maka a*b > $\sqrt{c}$ * $\sqrt{c}$ = c, yang merupakan kontradiksi.

Lemma 3.2
Pada saat kita mau mengelimininasi semua kelipatan I, kita dapat memulai loop dari I*I karena semua kelipatan I < I*I pasti sudah tereleminasi.

Bukti - Untuk semua kelipatan I < I*I, kita dapat menulisnya sebagai c*I, dimana c < I. Bila c adalah bilangan prima, maka semua kelipatan c sudah dieliminasi pada saat pemrosesan c dilakukan. Bila c bilangan komposit, maka ada sebuah faktor prima dari c yang $\le \sqrt{c}$. Karena faktor prima itu < I, maka semua kelipatan faktor prima tersebut sudah dieliminasi.

Pengembangan pertama untuk algoritma *Sieve of Eratosthenes* dapat diambil dari teorema 3.1. Perubahan dilakukan pada bagian 2, yaitu pada loop I. Loop I dapat dilakukan dari 2 sampai dengan $\lfloor \sqrt{N} \rfloor$ saja, karena setelah tahap tersebut semua I dimana A_I bernilai true pasti bilangan prima.

Pengembangan kedua dapat diambil dari lemma 3.2. Perubahan kali ini dilakukan pada loop J, dimana batas bawah perulangan diubah dari 2 menjadi I. Maka algoritma *Sieve of Eratosthenes* setelah dikembangkan menjadi :

```
Sieve-of-Eratosthenes-optimized(N)

for I ← 2 to N
    do A_I ← true

for I ← 2 to floor(sqrt(N))
    do if (A_I = true)
        then for J ← I to floor(N / I)
            do A_I*J ← false
→ A
```

Kompleksitas dari algoritma di atas masih bergantung pada loop bersarang I dan J, karena bagian tersebut menyumbang paling banyak proses. Untuk setiap bilangan prima I, loop J berjalan sebanyak O($\lfloor N/I \rfloor$ - I). Maka kompleksitas waktunya adalah O(N * ln ln N) dikurangi total semua bilangan prima $\le \sqrt{N}$. Total untuk X bilangan prima adalah X^2 ln X [4] dan jumlah bilangan prima antara 1 sampai N dapat diestimasi dengan N / ln N [5]. Bila kedua persamaan diatas digabungkan, didapat hasilnya secara asimtotik adalah O(N / ln N). Maka secara asimtotik, kompleksitas algoritma ini tetap O(N * ln ln N).

Kompleksitas memori untuk algoritma yang sudah dioptimisasi ini masih O(N), karena kedua optimisasi

yang diberikan bertujuan untuk memperbaiki konstanta pada kompleksitas waktu, bukan pada kompleksitas memori.

Untuk lebih memberikan gambaran mengenai peningkatan kinerja algoritma yang terjadi, di bawah ini diberikan tabel yang berisi total perulangan J yang dilakukan untuk versi 1(normal) dan versi 2(dioptimisasi).

Tabel 3.1
Perbandingan Proses Algoritma Versi 1 dan Versi 2

N	Versi 1	Versi 2	N * ln ln N
10^1	7	5	8,340
10^2	146	104	152,718
10^3	1958	1411	1932,645
10^4	23071	16981	22203,268
10^5	256808	193078	244347,036
10^6	2775210	2122048	2625791,914
10^7	29465738	22850051	27799425,943

Dari tabel di atas, terlihat bahwa kedua versi memiliki tingkat pertumbuhan yang cukup stabil bila dibandingkan dengan N * ln ln N, bahkan untuk ukuran N sama dengan 10.000.000. Versi kedua juga melakukan proses lebih sedikit untuk semua masukan, sekitar 0.75 dari versi 1. Karena itu, dapat disimpulkan bahwa kedua optimisasi di atas akan mempercepat jalannya algoritma sebesar konstanta c, dimana $0{,}7 \leq c < 1$.

Sesungguhnya masih banyak pengembangan yang dapat dilakukan untuk memperkecil konstanta dari algoritma, seperti loop j dibatasi pada bilangan ganjil saja, karena semua bilangan genap pasti sudah tercoret oleh prima 2. Cara ini dapat digeneralisasi dengan sebuah teknik yang disebut *Wheel Factorization*[6].

4. Aplikasi dalam *Problem Solving*

Pada dasarnya, semua permasalahan yang memerlukan daftar bilangan prima yang lebih kecil dari sebuah bilangan tertentu dapat diselesaikan dengan *Sieve of Eratosthenes*. Disini akan diberikan contoh dua buah permasalahan matematika yang dapat diselesaikan dengan algoritma *Sieve of Eratosthenes* yang diadaptasi sedikit, yaitu *Euler's phi function* dan *Euler's totient function*.

1. *Prime Counting Function*[7]

Prime counting *function* untuk x (selanjutnya dilambangkan dengan $\pi(x)$) adalah fungsi yang menyatakan banyaknya bilangan prima yang lebih kecil atau sama dengan x. $\pi(x)$ dapat didekati dengan memakai N / ln N, seperti yang disebutkan pada [5]. Sayangnya, estimasi tersebut baru akurat jika angka yang diberikan besarnya sudah mencapai orde 10^9. Bila tidak hasilnya melenceng terlalu besar. Apabila yang diperlukan adalah $\pi(x)$ yang tepat, maka mau tidak mau harus dicari semua bilangan prima yang terletak antara 1 sampai dengan x. Bagian ini dapat dicari dengan mudah memakai *Sieve of Eratosthenes.* Berikut ini adalah *pseudocode* untuk mencari $\pi(x)$.

```
Prime-counting-function(x)

total ← 0
for I ← 2 to x
    do A_I ← true

for I ← 2 to x
    do if (A_I = true)
        then total ← total + 1
                for J ← I to floor(x / I)
                    do A_I*J ← false
→ total
```

Keunggulan utama dari *pseudocode* diatas adalah kemudahannya untuk mengubahnya sehingga bisa menghitung semua $\pi(x)$ untuk x $\leq N$. nilai total saat ini hanya perlu disimpan ke dalam sebuah larik (sebut saja hasil) untuk setiap nilai I. Berikut ini adalah *pseudocode* generalisasinya

```
All-prime-counting-function(N)

total ← 0
for I ← 2 to N
    do A_I ← true

hasil_1 ← 0
for I ← 2 to N
    do if (A_I = true)
        then total ← total + 1
                for J ← I to floor(N / I)
                    do A_I*J ← false
        hasil_I ← total
→ hasil
```

Kompleksitas waktu dan memori dari algoritma di atas sama persis dengan algoritma *Sieve of Eratosthenes* versi 1, karena hanya ada penambahan jumlah operasi sebesar O(1) untuk setiap loop I dan penambahan memori sebesar O(1) untuk setiap nilai x $\leq N$.

2. *Euler's totient function*[8]

Euler's totient function (selanjutnya dilambangkan dengan $\phi(x)$) menyatakan jumlah bilangan bulat positif $< x$ yang relatif prima terhadap x. Karena pendefinisian

ini, maka untuk setiap bilangan prima p, $\phi(p) = p - 1$. Lebih lanjut lagi, untuk setiap bilangan yang merupakan hasil perpangkatan dari p, $\phi(p^k) = p^k * (1 - 1/p)$. Hal ini disebabkan untuk sebuah bilangan yang merupakan hasil perpangkatan dari p, bilangan-bilangan yang tidak relatif prima dengan bilangan itu adalah kelipatan p, dan total bilangan kelipatan p dalam rentang p^k adalah $p^{(k-1)}$.

Sifat lainnya yang penting untuk $\phi(x)$ adalah jika hasil faktorisasi prima x adalah 2 buah bilangan prima p dan q, maka $\phi(x) = x * (1 - 1/p) * (1 - 1/q)$. Hasil ini dapat digeneralisasi ke t prima, dimana hasilnya menjadi $\phi(x) = x * \prod_{i=1}^{t} (1 - 1/p_i)$, dimana p_t adalah hasil faktorisasi prima ke t dari x.

Dari persamaan di atas, terlihat bahwa untuk mencari $\phi(x)$, pertama-tama harus dicari semua faktor prima unik untuk x. Maka *Sieve of Eratosthenes* dapat dipakai untuk mencari semua faktor prima yang ada antara 1 sampai x, kemudian tinggal mencari bilangan prima yang membagi x. Berikut ini adalah *pseudocode* untuk menghitung $\phi(x)$.

```
Euler-totient-function(x)

total ← x
A ← Sieve-of-Eratosthenes(x)

for I ← 2 to x
    do if (A_I = true) and (x mod I = 0)
        then total ← total – total / I

→ total
```

Kompleksitas dari algoritma di atas adalah O(N ln ln N) untuk waktu dan O(N) untuk memori, karena proses diluar *Sieve of Eratosthenes* membutuhkan proses lebih sedikit, O(N) untuk waktu (looping I) dan O(1) untuk memori.

Sekarang andaikan yang diminta adalah menghitung semua $\phi(x)$ untuk $x \le N$. Kita bisa saja menggunakan cara berikut ini, hanya saja kompleksitas algoritma yang dihasilkan kurang bagus.

```
All-Euler-totient-function(N)

for I ← 1 to N
    do hasil_I ← Euler-totient-function(I)

→ hasil
```

Bahkan apabila kita memindahkan bagian pemanggilan *Sieve of Eratosthenes* ke luar dari Euler-totient-function, tetap saja hasilnya kurang baik, karena total kompleksitas waktunya tetap $O(N^2)$. Sekalipun fungsi Euler-totient-function diubah sehingga hanya perlu dilakukan perulangan sebanyak $\sqrt{x}$, tetap saja kompleksitasnya masih O(N * $\sqrt{N}$), terlalu besar apabila N yang diminta adalah 1.000.000.

Pengamatan yang diperlukan adalah pada saat kita melakukan *Sieve of Eratosthenes*, kita juga dapat langsung melakukan perhitungan untuk mendapatkan $\phi(x)$. Ini disebabkan pada loop J pada program *Sieve of Eratosthenes*, kita sudah pasti mendapatkan bilangan-bilangan yang merupakan kelipatan dari I. Maka cara yang tepat adalah kita melakukan perhitungan $\phi(x)$ sekaligus melakukan *Sieve of Eratosthenes*. Berikut adalah *pseudocode* yang menggambarkan caranya.

```
All-Euler-totient-function(N)

for I ← 1 to N
    do T_I ← I
       A_I ← true

for I ← 2 to N
    do if (A_I = true)
       then for J ← 2 to floor(N / I)
            do A_I*J ← false
               T_I*J ← T_I*J _ T_I*J / I
→ T
```

Dengan cara demikian, tidak ada proses percuma yang dilakukan, karena kita mencari kelipatan dari I yang dapat dilakukan secara efisien, bukan mencari pembagi dari sebuah bilangan yang dengan cara paling baik sekalipun masih lebih buruk dibandingkan dengan mencari kelipatan sebuah bilangan.

Komplesitas waktu dari algoritma di atas adalah O(N * ln ln N), karena jumlah proses yang ditambahkan adalah konstan per loop J. Kompleksitas memori dari algoritma ini juga tetap O(N) karena jumlah memori tambahan yang diperlukan adalah O(1) per elemen.

5. Kesimpulan

Sieve of Eratosthenes dapat mencari semua bilangan prima dalam rentang batas tertentu dengan kompleksitas waktu O(N * ln ln N), yang berarti pertumbuhan fungsinya nyaris ekuivalen dengan fungsi linier. Melihat dari sudut pandang apa yang dikerjakan, kompleksitas tersebut terasa sangat baik, terutama bila dibandingkan dengan *trial division*.

Kekurangan terbesar dari *Sieve of Eratosthenes* adalah kebutuhan memorinya yang besar, sehingga hanya dapat dipakai untuk N ≤ 10.000.000. Lebih dari itu, walaupun kompleksitas waktunya masih cukup baik, tetapi

kompleksitas memorinya tidak menunjang, karena jumlah memori yang diperlukan sudah berada dalam jenjang puluhan, bahkan ratusan Megabyte.

Apabila *Sieve of Eratosthenes* masih dirasa kurang cepat, bahkan setelah dioptimisasi, maka masih ada cara lain yang lebih cepat lagi, yaitu *Sieve of Atkin*, ditemukan pada tahun 2003 oleh A. O. L. Atkin dan D. J. Bernstein. Ide dari algoritma ini juga mengacu pada *Sieve of Eratosthenes*, menunjukkan bahwa ide dari *Sieve of Eratosthenes* sudah bagus.

6. Referensi

[1] http://en.wikipedia.org/wiki/Eratosthenes; Tanggal akses 10 Desember 2011.

[2] http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes; Tanggal akses 10 Desember 2011.

[3] http://mathworld.wolfram.com/HarmonicSeriesofPrimes.html; Tanggal akses 10 Desember 2011.

[4] http://mathworld.wolfram.com/PrimeSums.html; Tanggal akses 11 Desember 2011.

[5] Thomas H. Cormen et al., "Introduction to Algorithms, Second Edition", Massachusetts Institute of Technology, Cambridge, MA, Mei 2001. Hal. 888.

[6] http://en.wikipedia.org/wiki/Wheel_factorization; Tanggal akses 11 Desember 2011.

[7] http://mathworld.wolfram.com/PrimeCountingFunction.html; tanggal akses 11 Desember 2011.

[8] http://mathworld.wolfram.com/TotientFunction.html; tanggal akses 11 Desember 2011.

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, 11 Desember 2011

Christianto - 13510003