Algoritma Sublinear untuk Menghitung Banyak Bilangan Prima dengan Fenwick Tree

Kristo Anugrah - 13522024
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13522024@std.stei.itb.ac.id

***Abstrak*—Makalah ini membahas algoritma sublinear yang menggunakan struktur data *fenwick tree* dalam menghitung banyak bilangan prima yang lebih kecil dari sebuah bilangan bulat. *Fenwick tree* adalah struktur data yang memakai teknik *decrease and conquer* untuk menunjang operasi *update* dan *query*. Dengan mendefinisikan sebuah fungsi khusus, persoalan ini dapat diselesaikan dengan bantuan *fenwick tree*. Dengan tujuan mengoptimisasi algoritma, akan dipilih sebuah parameter khusus. Parameter ini akan menurunkan kompleksitas waktu algoritma secara drastis. Makalah ini juga akan melakukan perbandingan antara algoritma sublinear dengan algoritma penghitung bilangan prima lainnya, seperti *linear sieve* dan *sieve of eratosthenes*.**

***Kata kunci*—algoritma, bilangan prima, fenwick tree, optimisasi, sublinear**

I. Pendahuluan

Bilangan prima adalah bilangan bulat positif yang hanya dapat dibagi oleh 1 dan dirinya sendiri. Secara ekivalen, sebuah bilangan disebut prima jika hanya memiliki dua faktor. Bilangan asli 1 secara umum tidak dianggap prima.

Fungsi penghitung prima, $\pi(n)$, sama dengan banyak bilangan prima yang kurang dari atau sama dengan n. Contohnya, $\pi(2) = 1$, $\pi(5) = 3$, $\pi(10) = 4$. Walaupun $\pi(x)$ terdefinisi untuk x bilangan real, umumnya x merupakan bilangan bulat nonnegatif.

Untuk menghitung nilai $\pi(n)$ untuk bilangan n sembarang, diperlukan sebuah algoritma yang efisien. Algoritma naif seperti *trial division* yang memiliki kompleksitas $O(n\sqrt{n})$ membutuhkan waktu komputasi lama untuk bilangan n besar. Algoritma lain seperti *sieve of eratosthenes* memiliki kompleksitas waktu $O(n \log n)$ dan kompleksitas memori $O(n)$. Namun untuk n yang besar algoritma ini masih membutuhkan waktu dan memori yang relatif besar (untuk $n = 10^{11}$, *sieve of eratosthenes* membutuhkan 400 GB memori!).

Fenwick tree atau *binary indexed tree* merupakan sebuah struktur data yang dapat melakukan operasi *prefix sum* dan *update* nilai elemen *array* secara efisien. Struktur data ini dikemukakan oleh Boris Ryabko pada tahun 1989 dengan modifikasi tambahan pada tahun 1992. Untuk sebuah *array*, operasi *prefix sum* dapat dilakukan secara naif dalam $O(n)$, sedangkan operasi *update* dapat dilakukan dalam $O(1)$. Struktur data *fenwick tree* memungkinkan kedua operasi untuk dijalankan dalam $O(logn)$. Hal ini dicapai dengan merepresentasikan *array* linear dengan n elemen sebagai sebuah pohon dengan $n + 1$ simpul.

Dalam makalah ini, proses penghitungan banyak bilangan prima akan dipercepat dengan mengunakan struktur data *fenwick tree*. Selain itu, akan didefinisikan fungsi δ (delta) yang akan membuat perhitungan makin efisien. Proses optimisasi ini akan menghasilkan algoritma dengan kompleksitas waktu $O(n^{2/3} \log^{1/3} n)$.

II. Dasar Teori

A. Prime-counting Function

Prime-counting function adalah fungsi penghitung banyaknya bilangan prima yang kurang dari atau sama dengan suatu bilangan real x. *Prime-counting function* dari suatu bilangan real x dilambangkan dengan $\pi(x)$.

Misal p_n melambangkan bilangan prima ke-n, maka p_n adalah *right inverse* dari $\pi(x)$, karena $\pi(p_n) = n$ untuk seluruh bilangan bulat positif.

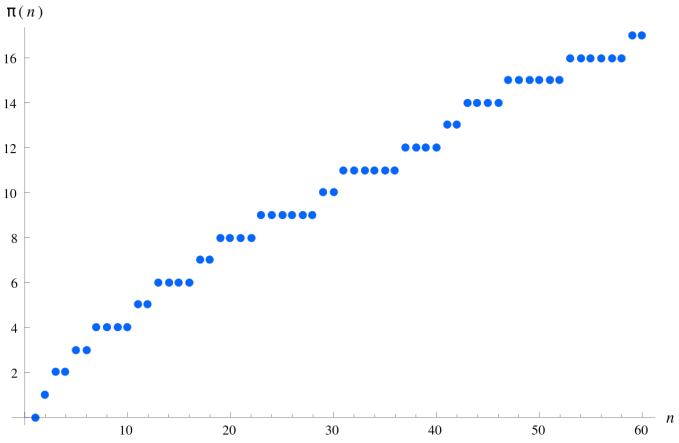

Fig. 1. Nilai dari $\pi(n)$ untuk $n \leq 60$, dari ResearchGate [2024].

B. Teorema Bilangan Prima

Teorema bilangan prima mendeskripsikan distribusi asimtotik bilangan prima di antara bilangan bulat positif. Teorema bilangan prima menjelaskan fakta bahwa bilangan prima

menjadi semakin jarang ditemukan di antara bilangan besar. Teorema ini mengukur laju kemunculan bilangan prima di antara bilangan nonprima.

Teorema bilangan prima menyatakan bahwa:

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\log(x)} = 1, \tag{1}$$

dengan $\log(x)$ melambangkan logaritma natural dari x. Atau, secara ekivalen, untuk bilangan x yang cukup besar, $\pi(x) \approx x/\log(x)$. Perhatikan bahwa teorema ini tidak menyatakan batas perbedaan antara $\pi(x)$ dan $x/\log(x)$. Teorema ini hanya menyatakan bahwa besar *error* relatif terhadap x mendekati 0 untuk x yang membesar tanpa batas.

Secara ekivalen, teorema bilangan prima juga menyatakan:

$$\lim_{x\to\infty} \frac{\pi(x)}{\mathrm{Li}(x)} = 1, \tag{2}$$

dimana $\mathrm{Li}(x)$ melambangkan fungsi integral logaritmik yang didefinisikan sebagai:

$$\mathrm{Li}(x) = \int_2^x \frac{\mathrm{d}t}{\ln t}. \tag{3}$$

Perhatikan bahwa meskipun $x/\log(x)$ dan $\mathrm{Li}(x)$ keduanya merupakan aproksimasi dari $\pi(x)$, $\mathrm{Li}(x)$ merupakan aproksimasi yang jauh lebih baik. Secara sederhana, hal ini karena untuk $x > 10$, berlaku $\pi(x) > x/\log(x)$, sedangkan $\pi(x) - \mathrm{Li}(x)$ berganti tanda tak hingga kali. Artinya, $x/\log(x)$ selalu meng-*underestimate* nilai $\pi(x)$, sedangkan $\mathrm{Li}(x)$ tidak meng-*overestimate* maupun meng-*underestimate* nilai $\pi(x)$. Secara formal, fakta tersebut dapat ditulis sebagai:

$$\pi(x) - \mathrm{Li}(x) = O(x^{1/2}\log x), \tag{4}$$

$$\pi(x) - x/\log x = O(x/\log^2 x). \tag{5}$$

Lebih jelasnya, berikut ini merupakan nilai dari $\pi(x)$, $\mathrm{Li}(x)$, serta $x/\log x$ serta nilai *relative error*-nya.

TABEL I
PERBANDINGAN APROKSIMASI DARI $\pi(x)$, DIADAPTASI DARI OEIS.

x	$\pi(x)$	$\pi(x) - x/\log x$	$\mathrm{Li}(x) - \pi(x)$	$x/\log x$ error %	$\mathrm{Li}(x)$ error %
10	4	0	2	8,22%	42,606%
10^2	25	3	5	14,06%	18,597%
10^3	168	23	10	14,85%	5.561%
10^4	1.229	143	17	12,37%	1,384%
10^5	9.592	906	38	9,91%	0,393%
10^6	78.498	6.116	130	8,11%	0,164%
10^7	664.579	44.158	339	6,87%	0,051%
10^8	5.761.455	332.774	754	5,94%	0,013%

C. Teorema Dasar Aritmetika

Teorema dasar aritmetika (juga disebut teorema faktorisasi unik) adalah sbeuah teorema *number theory*. Teorema ini menyatakan bahwa setiap bilangan bulat positif yang lebih besar dari 1 dapat ditulis sebagai hasil perkalian bilangan prima (atau bilangan bulat itu sendiri adalah bilangan prima). Teorema ini juga mengatakan bahwa hanya ada satu cara untuk menulis bilangan tersebut.

Secara formal, teorema ini menyatakan bahwa:

$$n = p_1^{n_1} p_2^{n_2} \dots p_k^{n_k} = \prod_{i=1}^{k} p_i^{n_i}, \tag{6}$$

berlaku untuk seluruh bilangan bulat positif $n > 1$, dimana $p_1 < p_2 < \dots < p_k$ merupakan bilangan prima dan n_i merupakan bilangan bulat positif. Bentuk tersebut juga disebut sebagai *canonical form* atau *standard form* dari n. Contohnya:

- $1000 = 2^3 \times 5^3$
- $1001 = 7 \times 11 \times 13$
- $1002 = 2 \times 3 \times 167$

Teorema ini dapat digeneralisasi untuk merepresentasikan seluruh bilangan bulat positif n:

$$n = 2^{n_1} 3^{n_2} 5^{n_3} 7^{n_4} \dots = \prod_{i=1}^{\infty} p_i^{n_i}, \tag{7}$$

dengan banyak n_i positif yang finit, dan sisanya bernilai 0.

D. Fenwick Tree

Fenwick tree merupakan struktur data yang dapat melakukan operasi *update* dan *query prefix sum* dari suatu *array* linier dalam $O(\log N)$. *Fenwick tree* merupakan struktur data yang memanfaatkan teknik *decrease and conquer* dalam implementasinya. Secara formal, jika diberikan suatu array A dengan panjang N, *fenwick tree* dapat menunjang operasi:

- *Update*: ubah satu nilai elemen dari array A. Lebih lengkapnya, diberikan bilangan bulat $0 \le i < N$ serta suatu nilai C, lakukan $A_i = C$.
- *Query*: hitung jumlah elemen dari awal hingga suatu indeks i. Lebih lengkapnya, diberikan bilangan bulat $0 \le i < N$, hitung $\sum_{j=0}^{i} A_j$. Operasi ini dapat digunakan untuk menunjang operasi lain yang lebih kompleks, seperti operasi *range sum query*, dimana diminta untuk menjumlahkan elemen array A dari indeks i hingga j.

Secara umum, struktur data *fenwick tree* dapat melakukan nilai dari fungsi f untuk sebuah *range* $[l, r]$. Namun, dalam kasus ini f dimisalkan sebagai fungsi penjumlahan ($f(l, r) = \sum_{j=l}^{r} A_j$).

Sebuah *fenwick tree* dari *array* A adalah *array* $T[0...N-1]$, dengan setiap elemennya merupakan jumlah elemen dari A untuk suatu *range* $[g(i), i]$:

$$T_i = \sum_{j=g(i)}^{i} A_j, \tag{8}$$

dengan $g(i)$ merupakan suatu fungsi yang memenuhi $0 \leq g(i) \leq i$. Definisi fungsi g akan dijelaskan nanti.

Dengan definisi di atas, maka operasi *query* dapat dilakukan dengan *pseudocode* berikut:

Algoritma 1 Algoritma operasi query

```
function QUERY(i)
    sum ← 0
    while i ≥ 0 do
        sum ← sum + T_i
        i ← g(i) − 1
    end while
    → sum
end function
```

Sedangkan *pseudocode* untuk operasi *update* adalah sebagai berikut:

Algoritma 2 Algoritma operasi update

```
procedure UPDATE(i, C)
    diff ← C − T_i
    for j yang memenuhi g(j) ≤ i ≤ j do
        T_j ← T_j + diff
    end for
end procedure
```

Jelas kompleksitas algoritma bergantung dari definisi fungsi g. Definisi fungsi g harus memenuhi $0 \leq g(i) \leq i$ untuk seluruh i. Akan didefinisikan fungsi g sedemikian sehingga operasi *update* dan *query* memiliki kompleksitas $O(\log N)$.

Fungsi $g(i)$ dideskripsikan dengan mengganti seluruh *trailing* 1 menjadi 0. Contohnya:

- $g(11) = g(1011_2) = 1000_2 = 8$
- $g(12) = g(1100_2) = 1100_2 = 12$
- $g(13) = g(1101_2) = 1100_2 = 12$
- $g(14) = g(1110_2) = 1110_2 = 14$
- $g(15) = g(1111_2) = 0000_2 = 0$

Untuk menghitung nilai $g(i)$, hanya diperlukan untuk melakukan bitwise AND dengan i dan i+1. Secara formal, fungsi g dihitung dengan:

$$g(i) = i \ \& \ (i+1), \tag{9}$$

dengan & menandakan operasi bitwise AND.

Untuk operasi *query*, diperlukan suatu fungsi yang memungkinkan kita untuk mencari seluruh j yang memenuhi $g(j) \leq i \leq j$. Akan didefinisikan fungsi h untuk tujuan tersebut.

Mudah dilihat bahwa seluruh j dapat diperoleh dengan memulai dari i dan menginversi bit terakhir yang bernilai 0. Contohnya, untuk $i = 10 = 0001010_2$ didapat urutan operasi:

1) $h(10) = h(0001010_2) = 0001011_2 = 11$
2) $h(11) = h(0001011_2) = 0001111_2 = 15$
3) $h(15) = h(0001111_2) = 0011111_2 = 31$
4) $h(31) = h(0011111_2) = 0111111_2 = 63$
5) dan seterusnya hingga batas atas.

Ternyata, fungsi h juga memiliki proses kalkulasi yang relatif mudah. Fungsi h didefinisikan dengan:

$$h(i) = i \parallel (i+1), \tag{10}$$

dengan $\parallel$ melambangkan operasi bitwise OR.

Perlu diiterasikan bahwa meskipun struktur data ini bernama *fenwick tree*, dalam implementasinya struktur data ini merupakan sebuah *array*. Aspek ”pohon” dari struktur data ini muncul karena terdapat representasi struktur data ini sebagai pohon. Meskipun begitu, kita tidak perlu memodelkan struktur data sebenarnya sebagai kumpulan simpul dan sisi.

Dalam representasi pohon, simpul i memiliki *parent* simpul $i \parallel (i+1)$. Representasi ini mengartikan sebuah simpul bisa memiliki banyak anak.

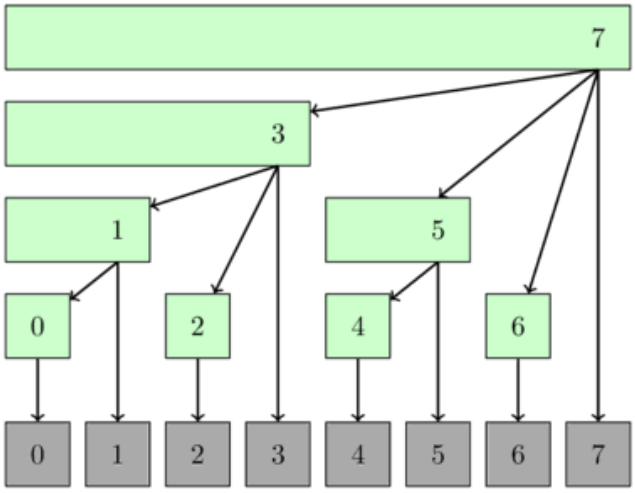

Fig. 2. Representasi pohon *fenwick tree*, dari cp-algorithms [2024].

Perhatikan bahwa definisi fungsi g dan h memungkinkan operasi *update* dan *query* untuk dijalankan dalam waktu $O(\log n)$. Untuk fungsi g, karena setiap operasi setidaknya membuat 1 bit yang awalnya 1 menjadi 0, pada akhirnya seluruh bilangan bulat positif i akan berakhir di angka 0. Banyak bit 1 yang dapat dimiliki oleh suatu bilangan bulat positif adalah $O(\log n)$. Untuk fungsi h, hal sebaliknya terjadi. Setiap operasi h setidaknya membuat 1 bit yang awalnya 0 menjadi 1. Banyak bit 0 yang dapat suatu bilangan bulat positif miliki adalah $O(\log n)$.

E. Linear Sieve

Diberikan sebuah bilangan bulat positif n, algoritma *linear sieve* akan mencari faktor prima terkecil untuk setiap bilangan $i \in [2; n]$ dalam waktu $O(n)$. Hasil faktor prima terkecil untuk setiap bilangan akan disimpan di dalam sebuah *array* lp. Selain itu, selama berjalannya algoritma dibutuhkan *list* kumpulan bilangan prima yang telah ditemukan - yaitu *list* pr. Awalnya, seluruh index pada *array* lp akan diinisialisasi dengan nilai 0, yang artinya kita mengasumsikan seluruh bilangan adalah bilangan prima.

Selama berjalannya algoritma, ada 2 kasus yang dapat terjadi:

- $lp[i] = 0$ – yang artinya bilangan i adalah bilangan prima, karena kita tidak menemukan faktor prima yang lebih kecil untuk bilangan tersebut. Oleh karena itu, dilakukan *assignment* $lp[i] = i$ dan nilai i akan dimasukkan ke dalam *list* pr.
- $lp[i] \neq 0$ – yang artinya bilangan i bukan bilangan prima, dan faktor prima terkecilnya adalah $lp[i]$.

Perhatikan bahwa setiap bilangan bulat positif dapat direpresentasikan sebagai $i = lp[i] \times x$, dimana $lp[i]$ adalah faktor prima terkecil dari i, dan bilangan x tidak memiliki faktor prima yang lebih kecil dari $lp[i]$, yaitu $lp[i] \leq lp[x]$.

Dengan observasi tersebut, untuk setiap bilangan i kita akan memperbarui setiap $x_j = i \times p_j$, dengan p_j melambangkan seluruh bilangan prima yang kurang dari atau sama dengan $lp[i]$. Untuk setiap x_j, akan dilakukan *update* $lp[x_j] = p_j$. *Pseudocode* dari *linear sieve* adalah sebagai berikut:

Algoritma 3 Algoritma linear sieve

```
1:  function LINEARSIEVE(n)
2:      pr ← []
3:      lp ← [0..n − 1]
4:      for i ← 2 to n do
5:          if lp[i] = 0 then
6:              lp[i] ← i
7:              pr ← pr ∪ i
8:          end if
9:          j ← 0
10:         stillMinimum ← true
11:         while i × pr[j] ≤ n and stillMinimum do
12:             lp[i × pr[j]] ← pr[j]
13:             if pr[j] = lp[i] then
14:                 stillMinimum ← false
15:             end if
16:             j ← j + 1
17:         end while
18:     end for
19:     → lp
20: end function
```

III. Analisis dan Implementasi

A. Definisi

Sebelum melanjutkan ke perancangan solusi, ada beberapa definisi yang akan digunakan:

- $\pi(n)$ — fungsi *prime counting function*, yaitu banyak bilangan prima yang tidak lebih dari n.
- p_i — bilangan prima ke-i.
- $\delta(n, a)$ — jumlah bilangan bulat dalam $[1; n]$ yang tidak dapat dibagi oleh a bilangan prima pertama. Secara definisi, fungsi δ selalu menghitung 1, karena 1 tidak dapat dibagi oleh bilangan prima apapun. Perlu diperhatikan bahwa fungsi ini tidak menghitung bilangan prima yang kurang dari n.

Pertama-tama, akan didefinisikan fungsi δ. Perhatikan bahwa $\delta(n, a)$ akan menghitung seluruh bilangan positif $1 \leq i \leq n$ dimana faktor prima terkecil dari $i > p_a$. Untuk menghitung nilai $\delta(n, a)$, kita perlu mempertimbangkan kontribusi dari $\delta(n, a-1)$. Mudah dilihat bahwa nilai dari $\delta(n, a)$ yaitu nilai $\delta(n, a-1)$ yang dikurangi dengan seluruh bilangan positif kurang dari n yang memiliki faktor prima terkecil p_a. Secara formal:

$$\delta(n, a) = \delta(n, a-1) - |P|, \tag{11}$$

dengan P merupakan himpunan bilangan bulat positif x kurang dari n sehingga faktor prima terkecil dari x adalah p_a.

Perhatikan bahwa $\delta(\lfloor n/p_a \rfloor, a-1)$ memiliki korespondensi satu-satu dengan himpunan P. Hal ini karena bilangan yang dihitung pada $\delta(\lfloor n/p_a \rfloor, a-1)$ dapat dikali dengan p_a yang merupakan elemen P. Seluruh bilangan yang ada pada $\delta(\lfloor n/p_a \rfloor, a-1)$ memiliki faktor prima terkecil lebih dari p_a. Apabila bilangan tersebut dikali dengan p_a, faktor prima terkecilnya pastilah sama dengan p_a. Atau, secara formal:

$$|P| = \delta(\lfloor n/p_a \rfloor, a-1). \tag{12}$$

Dengan observasi tersebut, persamaan 11 dapat diubah menjadi:

$$\delta(n, a) = \delta(n, a-1) - \delta(\lfloor n/p_a \rfloor, a-1). \tag{13}$$

B. Rancangan Solusi

Akan diambil sebuah bilangan bulat positif y yang memenuhi $\sqrt[3]{n} \leq y \leq \sqrt{n}$. Konstanta y akan digunakan untuk mempercepat proses penghitungan.

Dengan konstanta y, akan didefinisikan hubungan fungsi π dengan fungsi δ. Fungsi $\pi(n)$ dapat ditulis sebagai:

$$\pi(n) = \delta(n, \pi(y)) + \pi(y) - F - 1, \tag{14}$$

dengan F merupakan banyak bilangan bulat positif di rentang $[1; n]$ yang dapat diekspresikan sebagai pq dengan $y < p \leq q$ dan p, q bilangan prima. Intuisi dari persamaan 14 adalah sebagai berikut:

1) $\delta(n, \pi(y))$ — nilai $\delta(n, \pi(y))$ merepresentasikan seluruh bilangan bulat positif dalam rentang $[1; n]$ dengan faktor prima terkecil yang melebihi $p_{\pi(y)}$. Perhatikan bahwa suku ini tidak mengitung banyak bilangan prima yang kurang dari atau sama dengan $p_{\pi(y)}$.
2) $\pi(y)$ — karena suku (1) tidak menghitung banyak bilangan prima yang kurang dari $p_{\pi(y)}$, suku ini akan digunakan untuk menambahkan banyak bilangan prima tersebut ke hasil.
3) F — merupakan banyak bilangan bulat positif di rentang $[1; n]$ yang dapat diekspresikan sebagai perkalian bilangan prima p dan q dengan $y < p \leq q$. Suku ini dibutuhkan karena suku (1) dan (2) masih memuat bilangan yang memiliki faktor prima terkecil lebih dari y. Oleh karena itu, bilangan tersebut perlu dikurangi dari hasil.

Perhatikan bahwa bilangan yang memenuhi kriteria tersebut pasti merupakan perkalian dari dua bilangan prima. Karena $y > \sqrt[3]{n}$, tidaklah mungkin bilangan tersebut memiliki lebih dari 2 faktor prima, karena $y \times y \times y > n$.

4) -1 — karena suku (1), (2), dan (3) masih menghitung angka 1, hasil tersebut perlu dikurangi 1.

Dengan definisi δ sesuai dengan persamaan 14, nilai $\delta(n, \pi(y))$ nantinya akan dihitung dengan metode rekursif. Namun, kita akan menghentikan proses rekursif ketika menghitung $\delta(m, a)$ dengan $m \leq n/y$. Ini artinya bentuk fungsi lainnya akan memenuhi bentuk $\delta(\lfloor n/k \rfloor, a)$, dengan $k < y$ dan $a \leq \pi(y)$. Ini artinya proses kalkulasi akan memiliki kompleksitas waktu $O(y\pi(y))$, yang dapat disederhanakan menjadi $O(y^2/\log y)$ dengan distribusi bilangan prima. Namun, karena y konstan dan terdefinisi menurut n, kompleksitas waktu tersebut dapat ditulis sebagai $O(y^2/\log n)$.

Observasi lain yang penting adalah dalam menghitung nilai dari fungsi δ, kita tidak perlu secara langsung menghitung nilai tersebut. Bentuk rekursif dari persamaan 13 memungkinkan kita untuk menghitung nilai fungsi δ di akhir. Lebih tepatnya, ketika menghitung nilai δ kita hanya perlu menyimpan *tuple*$(m, a, sign)$, dengan $sign \in \{-1, 1\}$ dan $m \leq n/y$. Hasil akhirnya kemudian hanya perlu ditambahkan dengan $\delta(m, a) \times sign$.

Dengan observasi tersebut, kita akan menghitung nilai δ dengan struktur data *fenwick tree*. Dengan melakukan *linear sieve* kita akan mendapatkan faktor prima terkecil untuk setiap bilangan bulat positif dalam rentang $[1; n/y]$. Faktor prima tersebut dapat disimpan dalam sebuah *array*. Untuk menghitung nilai dari $\delta(n, a)$, hanya perlu dihitung jumlah bilangan bulat $i \in [1; m]$ sedemikian sehingga faktor prima terkecil i lebih besar dari p_a.

Kalkulasi ini dapat dilakukan dengan *fenwick tree* dengan memodelkan setiap faktor prima terkecil dari setiap bilangan di rentang $[1; n/y]$ sebagai sebuah indeks dalam suatu *array*. Ketika kita ingin mengkueri nilai dari $\delta(m, a)$, hanya perlu dilakukan operasi *query* pada *fenwick tree* dengan indeks kueri a, misal hasil kueri c. Nilai dari $\delta(m, a)$ adalah $m - c$.

Operasi di atas dapat dilakukan dalam kompleksitas waktu $O(n/y \log n/y) = O(n/y \log n)$ dengan melakukan kueri setelah menyimpan seluruh bentuk rekursif dan menyortir *tuple* $(m, a, sign)$ terhadap nilai m terlebih dahulu. Hal ini karena dalam kasus terburuk, untuk setiap bilangan bulat positif di rentang $[1; n/y]$ akan dilakukan kueri terhadap *fenwick tree* yang memiliki kompleksitas waktu $O(\log n)$.

Penghitungan nilai F dapat dilakukan dengan menggunakan *list* bilangan prima yang dihasilkan dari algoritma *linear sieve*. Setiap bilangan prima dalam *list* tersebut yang lebih dari n/y akan diiterasi satu per satu. Untuk setiap bilangan prima, akan dicari indeks bilangan prima terkecil j sedemikian sehingga $p_i \times p_j \leq n$. Proses ini dapat dilakukan dengan menyimpan *pointer* lain yang menandakan nilai j yang sesuai. Perhatikan bahwa untuk indeks i yang menaik, j akan selalu menurun, sehingga proses ini bersifat linear.

Secara garis besar, berikut ini adalah langkah-langkah algoritma:

1) Tetapkan nilai y dari input n yang diberikan. Nilai y harus memenuhi $\sqrt[3]{n} \leq y \leq \sqrt{n}$.
2) Lakukan *linear sieve* untuk menemukan nilai dari $\pi(y)$. Proses *linear sieve* akan menggunakan nilai n/y sebagai batas atas yang harus dikomputasi. Proses *linear sieve* ini akan menghasilkan *array* lp dan pr, yaitu *array* yang berisi faktor prima terkecil dan *list* bilangan prima yang kurang dari n/y.
3) Dari *list* bilangan prima yang dihasilkan *linear sieve*, hitung nilai F. Proses ini secara garis besar akan mengiterasi *list* bilangan prima satu per satu dan kemudian mencari indeks batas bilangan prima yang jika dikali hasilnya kurang dari n.
4) Hitung nilai $\delta(n, \pi(y))$ dengan menggunakan *fenwick tree*. Proses ini akan menggunakan *array* lp yang dihasilkan oleh proses *linear sieve*. Secara garis besar, *tuple* $(m, a, sign)$ akan disortir dengan m membesar, yang kemudian hasilnya dihitung menggunakan *fenwick tree*.

C. Analisis Kompleksitas

Dari rancangan solusi yang telah dibahas, jelas bahwa algoritma memiliki kompleksitas waktu total:

$$\text{Kompleksitas waktu} = O\left(\frac{y^2}{\log n}\right) + O\left(\frac{n}{y}\log n\right). \quad (15)$$

Jelas bahwa kompleksitas waktu algoritma bergantung dari pemilihan konstanta y. Diperlukan suatu konstanta y yang meminimalkan kompleksitas waktu tersebut. Kompleksitas waktu tersebut optimal ketika $y = n^{1/3}\log^{2/3} n$, yang akan menghasilkan kompleksitas waktu akhir:

$$\text{Kompleksitas waktu} = O(n^{2/3}\log^{1/3} n). \quad (16)$$

Perhatikan juga bahwa selama berjalannya algoritma, memori yang digunakan hanya berupa *fenwick tree* serta array lp dan pr yang digunakan pada proses *linear sieve*. Karena ketiga struktur data ini berbanding lurus terhadap batas atas yang diberikan, yaitu n/y, maka didapat kompleksitas ruang:

$$\text{Kompleksitas ruang} = O\left(\frac{n}{y}\right) = O\left(\frac{n^{2/3}}{\log^{1/3} n}\right). \quad (17)$$

IV. Studi Kasus

A. Uji Kebenaran Algoritma

Algoritma akan di-tes untuk beberapa nilai n yang akan dicocokkan dengan nilai $\pi(n)$ yang sebenarnya untuk memastikan kebenaran algoritma. Nilai n dalam uji ini memiliki representasi 10^n, dengan $n \in [1; 10]$. Nilai $\pi(n)$ diambil dari [1].

Dapat dilihat bahwa hasil algoritma dengan $\pi(n)$ memiliki nilai yang sama. Uji ini membuktikan bahwa algoritma benar dan tepat. Selain itu, dapat diobservasi bahwa algoritma berjalan dengan sangat cepat. Untuk $n = 10^{10}$, algoritma membutuhkan waktu kurang dari 1 detik. Waktu yang cepat ini membuktikan bahwa algoritma berjalan secara sublinear.

TABEL II
PERBANDINGAN HASIL ALGORITMA DENGAN NILAI $\pi(n)$

n	Hasil algoritma	$\pi(n)$	Waktu (ms)
10	4	4	0,037
10^2	25	25	0,031
10^3	168	168	0,039
10^4	1.229	1.229	0,056
10^5	9.592	9.592	0,278
10^6	78.498	78.498	0,781
10^7	664.579	664.579	2,031
10^8	5.761.455	5.761.455	9,044
10^9	50.847.534	50.847.534	39,394
10^{10}	455.052.511	455.052.511	174,738

B. Perbandingan dengan Algoritma Lain

Algoritma yang telah dirumuskan akan dibandingkan dengan algoritma lain seperti *sieve of eratosthenes* dan *linear sieve*.

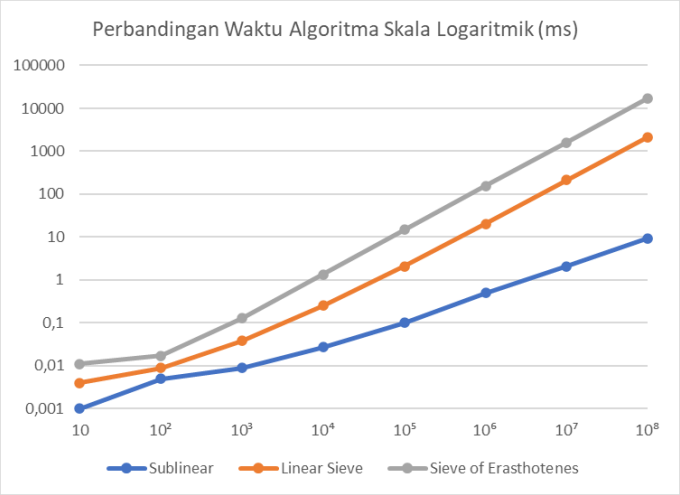

Fig. 3. Perbandingan algoritma sublinear dengan algoritma lain

Diketahui *sieve of eratosthenes* memiliki kompleksitas waktu $O(n \log n)$, *linear sieve* memiliki kompleksitas waktu $O(n)$, dan algoritma sublinear yang telah dibahas memiliki kompleksitas waktu $O(n^{2/3} \log^{1/3} n)$.

Dari figur di atas terlihat jelas bahwa algoritma sublinear merupakan peningkatan yang signifikan dibandingkan dengan algoritma *sieve of eratosthenes* dan *linear sieve*. Hal ini membuktikan kompleksitas waktu algoritma sublinear, karena algoritma berjalan jauh lebih cepat dari algoritma *linear sieve* yang memiliki kompleksitas waktu linear.

V. KESIMPULAN

Penggunaan struktur data *fenwick tree* dan optimisasi lainnya dapat mempercepat proses penghitungan fungsi $\pi(n)$, yaitu *prime-counting function*. Algoritma ini dibantu dengan penggunaan struktur data *fenwick tree*, yang menggunakan teknik *decrease and conquer*.

Algoritma ini memungkinkan proses kalkulasi yang jauh lebih cepat dibandingkan algoritma lain seperti *sieve of eratosthenes* dan *linear sieve*. Hal ini telah dibuktikan melalui waktu proses yang jauh lebih cepat dibanding algoritma lainnya. Diketahui algoritma ini memiliki kompleksitas waktu $O(n^{2/3} \log^{1/3} n)$ dan kompleksitas ruang $O(n^{2/3} / \log^{1/3} n)$.

VI. UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada:

1) Tuhan Yang Maha Esa,
2) orang tua penulis,
3) Bapak dan Ibu dosen pengampu mata kuliah Strategi Algoritma IF2211
4) teman-teman penulis, dan
5) pihak-pihak lain yang telah mendukung penulis selama pembelajaran dan proses pengerjaan makalah ini yang tidak dapat penulis sebutkan satu per satu.

REFERENSI

[1] Tomás Oliveira, S. (2007, March 28). Tables of values of PI(X) and of PI2(X). sweet.ua. Diakses 12 Juni 2024. https://sweet.ua.pt/tos/primes.html.

[2] Wikimedia. (n.d.). Prime_number_theorem_absolute_error. upload.wikimedia.org. Diakses 11 Juni 2024. https://upload.wikimedia.org/wikipedia/commons/thumb/6/6e/Prime_number_theorem_absolute_error.svg/300px-Prime_number_theorem_absolute_error.svg.png.

[3] cp-algorithms. (n.d.). binary_indexed_tree. cp-algorithms.com. Diakses 9 Juni 2024. https://cp-algorithms.com/data_structures/binary_indexed_tree.png.

[4] cp-algorithms. (25 November 2023). Linear Sieve. cp-algorithms.com. Diakses 11 Juni 2024. https://cp-algorithms.com/algebra/prime-sieve-linear.html.

[5] cp-algorithms. (19 April 2024). Fenwick Tree. cp-algorithms.com. Diakses 11 Juni 2024. https://cp-algorithms.com/data_structures/fenwick.html.

[6] Munir, R. (2024). Algoritma Decrease and Conquer (Bagian 2). informatika.stei.itb.ac.id. Diakses 10 Juni 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Decrease-and-Conquer-2024-Bagian2.pdf.

[7] Petersen, B. E. (1996). Prime Number Theorem. Department of Mathematics, Oregon State University. Diakses 9 Juni 2024. https://www.math.ucdavis.edu/~tracy/courses/math205A/PNT_Petersen.pdf.

[8] Lucy Hedgehog. (3 Mei 2013). Problem 10. projecteuler.net. https://projecteuler.net/thread=10;page=5.

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.

Kristo Anugrah 13522024