Analisis Kompleksitas Operasi pada *Segment Tree* dan *Fenwick Tree* dalam Menyelesaikan Persoalan *Range Sum Query*

Primanda Adyatma Hafiz 13520022
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13520022@std.stei.itb.ac.id

***Abstract*—Persoalan *Range Sum Query* (RSQ) merupakan persoalan klasik yang berkaitan dengan permrosesan *array* dan menerima *query* berupa *range index* kemudian ditampilkan total elemen pada *range index* tersebut. RSQ dapat diselesaikan dengan metode *brute force* ataupun *prefix sum* tetapi kurang efisien sehingga alternatifnya dengan menggunakan struktur data *tree* yaitu *Segment Tree* ataupun *Fenwick Tree*. Penggunaan struktur data tersebut akan meningkatkan efisiensi program dibandingkan dengan metode *brute force* dan setiap *query* dapat diproses dengan kompleksitas logaritmik.**

***Kata Kunci—Segment Tree, Fenwick Tree, Range Sum Query*, Pohon.**

I. PENDAHULUAN

Persoalan *Range Sum Query* (RSQ) adalah persoalan terkait dengan pemrosesan *array* yang menerima sejumlah *query* terdiri dari dua buah indeks *array* l dan r dengan $l \leq r$ kemudian ditampilkan hasil penjumlahan *array* dari indeks *l* hingga indeks *r*. Persoalan ini juga memungkinkan adanya *query* yang bertujuan untuk mengubah nilai suatu elemen di posisi tertentu pada *array* sehingga nantinya total dari suatu *range* pada *array* pun akan berubah pula.

Persoalan RSQ merupakan sebuah persoalan klasik di dunia permograman. Soal yang berkaitan dengan RSQ banyak ditemui pada soal-soal mengenai algoritma dan struktur data. Biasanya permasalahan ini menerima masukan *query* dengan jumlah yang cukup banyak sehingga perlu dipilih metode terbaik agar waktu yang dibutuhkan menjadi minimum. Terdapat beberapa alternatif solusi untuk menyelesaikan persoalan ini di antaranya yaitu dengan metode *brute force*, *prefix sum*, *segment tree*, dan *fenwick tree*.

Metode *brute force* adalah sebuah pendekatan yang sangat jelas untuk memecahkan persoalan, biasanya penyelesaian dengan metode ini hanya didasarkan pada *problem statement* dan definisi konsep yang dilibatkan. Dalam hal persoalan RSQ, metode ini akan memproses tiap *query* yang masuk dengan menghitung secara satu per satu total dari setiap elemen pada suatu *range array*. Oleh karena itu, metode ini terbukti kurang efektif untuk menyelesaikan persoalan RSQ karena perhitungan jumlah elemen dilakukan satu per satu.

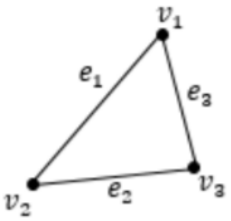
Selain metode *brute force*, terdapat pula metode *prefix sum* yang memanfaatkan *array* tambahan untuk menyimpan hasil penjumlahan elemen-elemen *prefix* dari suatu indeks *array*. Pada metode ini, posisi *array* ke-*i* akan berisi penjumlahan elemen dari indeks ke-1 hingga ke-*i* secara inklusif. Metode ini sangat efektif untuk menghitung total nilai pada suatu *range array*, akan tetapi jika elemen pada *array* nilainya dapat berubah maka algoritma *prefix sum* akan menjadi kurang efektif karena pada kasus terburuknya seluruh elemen pada *array* tambahan perlu diubah nilainya.

Oleh karena itu, pada makalah ini penulis akan membahas dua alternatif penyelesaian persoalan RSQ lainnya yaitu dengan menggunakan *Segment Tree* serta *Fenwick Tree*. Penulis juga akan menganalisis efisiensi program dengan menggunakan dua metode tersebut serta operasi-operasi dasar yang diperlukan pada setiap metode.

II. LANDASAN TEORI

A. Graf

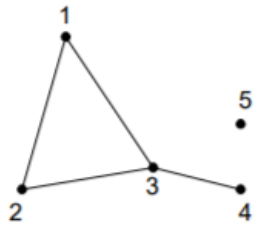
Graf adalah pasangan dua himpunan yaitu V himpunan tak kosong disebut simpul dan E yang anggotanya disebut sebagai sisi. Sebagai contoh, terdapat graf $G = (V, E)$ yang memiliki $V = \{v_1, v_2, v_3\}$ dan $E = \{(v_1, v_2), (v_2, v_3), (v_1, v_3)\}$ dapat direpresentasikan sebagai berikut.



Gambar 1. Representasi Graf G

B. Sirkuit

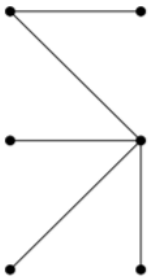
Sirkuit adalah lintasan yang berawal dan berakhir pada simpul yang sama. Sebagai contoh sebuah graf G yang memiliki sirkuit pada lintasan 1, 2, 3, 1 adalah sebagai berikut.



Gambar 2. Contoh Sirkuit
Diambil dari
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf pada 5 Desember 2021

C. Pohon

Pohon adalah graf tak terarah terhubung yang tidak mengandung sirkuit. Oleh karena itu, pohon tergolong sebagai graf akan tetapi graf belum tentu merupakan pohon. Adapun contoh representasi dari pohon adalah sebagai berikut.



Gambar 3. Contoh Pohon
Diambil dari
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf pada 5 Desember 2021

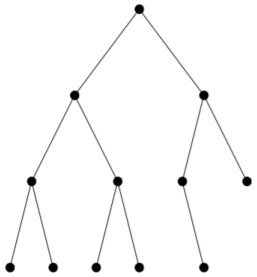
D. Sifat Pohon

Misalkan $G = (V, E)$ adalah graf tak berarah sederhana dan jumlah simpulnya n. G memiliki sifat-sifat sebagai berikut :

1. G adalah pohon.
2. Setiap pasang simpul di dalam G terhubung dengan lintasan tunggal.
3. G terhubung dan memiliki $m = n - 1$ buah sisi.
4. G tidak mengandung sirkuit dan memiliki $m = n - 1$ buah sisi
5. G tidak mengandung sirkuit dan penambahan satu sisi pada graf akan membuat hanya satu sirkuit.
6. G terhubung dan semua sisinya adalah jembatan.

E. Pohon Biner

Pohon biner adalah pohon n-ary dengan $n = 2$. Setiap simpul pada pohon biner terdiri atas paling banyak dua buah anak yang dibedakan menjadi anak kiri (*left child*) dan anak kanan (*right child*). Pohon biner tergolong sebagai pohon terurut karena urutan anak akan mempengaruhi representasi dari pohon biner. Contoh representasi dari pohon biner adalah sebagai berikut.



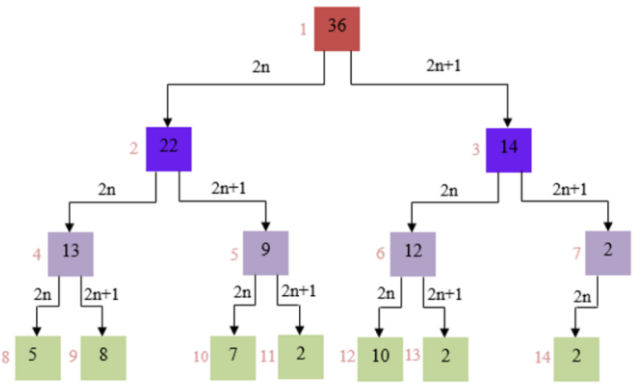
Gambar 4. Contoh Pohon Biner
Diambil dari
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag2.pdf pada 5 Desember 2021

F. Segment Tree

Segment Tree adalah sebuah tipe struktur data yag pertama kali diperkenalkan oleh Bentley pada tahun 1977. Pada prinsipnya *Segment Tree* adalah sebuah pohon biner yang setiap simpulnya menyimpan informasi mengenai sebuah segmen kontigu dari sebuah data struktur linier seperti *array*. *Segment Tree* adalah strukur data yang dapat menjawab sejumlah *range query* meskipun elemen pada *array* nilainya diubah-ubah. Operasi-operasi dasar pada *Segment Tree* adalah sebagai berikut.

1. Pembentukan *tree*
   Pada tahap ini akan dibuat struktur dari *Segment Tree* dengan memanfaatkan struktur data *array*.
2. Pengubahan nilai pada *array*
   Segment Tree memungkinkan pengubahan nilai pada posisi tertentu di *array* secara efisien. Pada tahapan ini simpul pada *Segment Tree* yang memuat informasi dari posisi *array* yang diubah akan diganti pula nilainya.
3. Penjawaban *query*
   Dari setiap *query* yang diterima akan dicari simpul yang berasosiasi dengan *range* pada *query* kemudian nilainya akan ditampilkan.

Pada persoalan RSQ setiap simpul pada *Segment Tree* akan menyimpan informasi *sum* dari suatu *range* indeks pada *array*. Contoh representasi *Segment Tree* dalam penyelesaian persoalan RSQ adalah sebagai berikut.



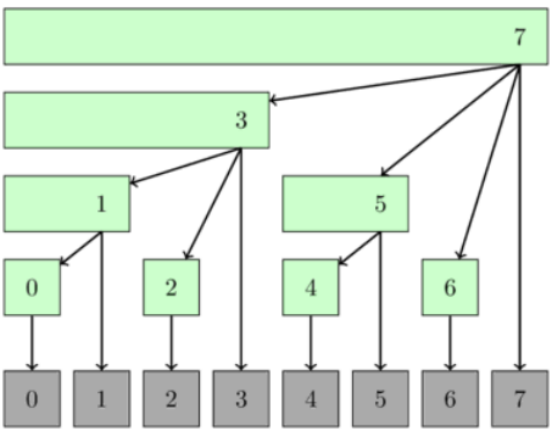
Gambar 4. Contoh Representasi Segment Tree
Diambil dari
https://www.baeldung.com/cs/segment-trees pada 5 Desember 2021

G. Fenwick Tree

Fenwick Tree atau sering disebut juga sebagai *Binary Indexed Tree* adalah struktur data yang pertama kali dikemukakakn oleh Boris Ryabko pada tahun 1989. Seperti halnya *Segment Tree*, *Fenwick Tree* juga merupakan pohon biner. Kegunaan dari keduanya sebenarnya mirip akan tetapi perbedaan *Fenwick Tree* dengan *Segment Tree* yaitu bahwa *Fenwick Tree* hanya membutuhkan memori sejumlah elemen *array* yang ingin dikalkulasi. Tiap simpul pada *Fenwick Tree* menyimpan informasi *range* dari suatu *array* dengan panjang dari setiap *range* sebesar 2^k dan $k \geq 0$. Jenis-jenis operasi dasar yang terdapat pada *Fenwcik Tree* adalah sebagai berikut.

1. Pembentukan *tree*
   Pada tahap ini akan dibuat struktur dari *Fenwick Tree* dengan memanfaatkan struktur data *array* yang besarnya sejumlah panjang $array$ yang ingin dikalkulasi.
2. Penambahan nilai pada *array*
   Pada tahapan ini simpul pada *Fenwick Tree* yang bit dari indeksnya berkorespondensi dengan indeks *array* yang nilainya mengalami penambahan akan ditambah pula nilainya.
3. Penjawaban *query*
   Dari setiap *range query* yang masuk akan dicari simpul-simpul yang indeksnya bersesuaian dengan mencocokkan representasi bit dari indeks pada *Fenwick Tree* kemudian nilainya akan dikembalikan.

Salah satu perbedaan mendasar antara *Fenwick Tree* dengan *Segment Tree* yaitu *Fenwick Tree* tidak dapat menyelesaikan persoalan *Range Maximum Query* dan *Range Minimum Query* karena properti dasar dari *Fenwick Tree* mirip dengan *Prefix Sum* yaitu mengkalkulasi nilai *range* $[l..r]$ yang memiliki sifat $pref(r) - pref(l-1)$. Oleh karena itu, *Fenwick Tree* sangat cocok untuk menyelesaikan persoalan RSQ. Adapun contoh representasi dari *Fenwick Tree* adalah sebagai berikut.



Gambar 5. Contoh Representasi Fenwick Tree
Diambil dari https://cp-algorithms.com/data_structures/fenwick.html pada 6 Desember 2021

H. Persoalan Range Sum Query (RSQ)

Pada prinsipnya persoalan RSQ merupakan persoalan yang bertujuan untuk mengkalkulasi total elemen dari sebuah *range* indeks pada *array*. Terdapat dua jenis *query* pada persoalan ini yaitu :

1. *Query* untuk kalkulasi *range sum*
   Query ini terdiri dari dua buah masukan yaitu l dan r kemudian akan dikembalikan nilai total dari elemen $arr[l..r]$.
2. *Query* untuk pengubahan/penambahan nilai
   Query ini terdiri dari dua buah masukan yaitu i dan x kemudian nilai dari elemen $arr[i]$ akan diubah sebesar x.

I. Kompleksitas Algoritma

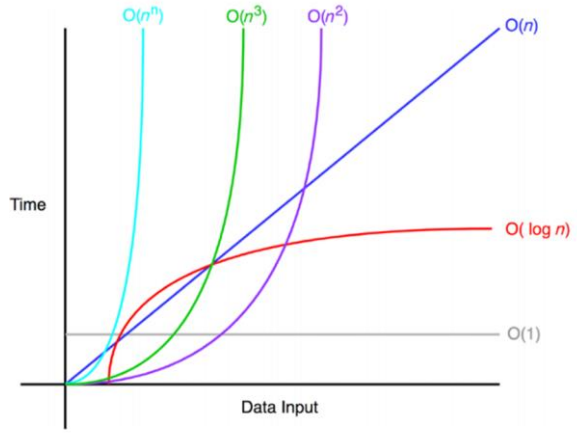
Dalam menyelesaikan persoalan komputasi algoritma tidak saja harus benar, tetapi juga harus efisien. Efisiensi dari suatu algoritma diukur dari jumlah waktu dan jumlah memori yang diperlukan oleh algoritma tersebut. Suatu algoritma dikatakan efisien bila meminimumkan kebutuhan waktu dan memori ketika program diimplementasikan.

Dalam menentukan kebutuhan waktu algoritma, mengukur langsung waktu yang dibutuhkan oleh suatu program agar dapat selesai berjalan bukanlah cara yang tepat. Hal ini dikarenakan setiap komputer memiliki spesifikasi dan arsitektur yang berbeda sehingga waktu setiap operasi antar komputer akan berbeda pula. Selain itu, bahasa pemorgraman yang berbeda juga akan menyebabkan waktu eksekusi program akan berbeda pula. Oleh karena itu dibutuhkan sebuah model yang dapat menghitung tingkat efektifitas algoritma yang independen terhadap jenis komputer dan *compiler* apapun.

Besaran yang dapat digunakan untuk menentukan efektifitas suatu algoritma dinamakan kompleksitas algoritma. Kompleksitas algoritma terbagi menjadi dua yaitu kompleksitas ruang dan kompleksitas waktu. Kompleksitas algoritma bekerja dengan mengukur laju pertumbuhan waktu/memori terhadap laju pertumbuhan masukan pada program. Salah satu notasi dalam menentukan kompleksitas algoritma yaitu notasi Big-O. Notasi Big-O menyatakan batas atas jumlah operasi primitif yang dilakukan program.

Terdapat beberapa jenis notasi Big-O antara lain yaitu O(1) yang menyatakan bahwa waktu eksekusi program tidak bergantung pada banyaknya data yang ada. Selain itu terdapat pula notasi Big-O polinomial seperti O(n), O(n^2), dan O(n^3) yang menyatakan bahwa waktu eksekusi program bergantung terhadap banyaknya data secara polinomial. Selanjutnya terdapat notasi Big-O logaritmik yaitu O($\log n$) yang menyatakan bahwa waktu eksekusi program bergantung pada banyaknya data yang ada secara logaritmik. Kemudian terakhir terdapat kompleksitas waktu eksponensial yang notasi Big-O nya dinyatakan dengan O(k^n) dengan $k > 1$.

Dari keseluruhan jenis kompleksitas di atas urutan tingkat efisiensi algoritma dari yang paling efisien hingga yang paling tidak efisien yaitu O(1), O($\log n$) O(n), O(n^2), O(n^3), O(k^n). Adapun perbandingan waktu dari kelimanya adalah sebagai berikut.



Gambar 6. Grafik Kompleksitas Algoritma
*Diambil dari https://ichi.pro/assets/images/max/724/1*iQkFjNn02ooqc2Yv27-pyQ.png pada 8 Desember 2021*

III. Analisis Kompleksitas pada *Segment Tree* dan *Fenwick Tree*

A. Segment Tree

1. Pembentukan *tree*

Pembentukan *tree* memanfaatkan strukur *array* yang indeksnya mengandung informasi level dari *node* serta *parent node*. Hal ini dimpelementasikan dengan memanfaatkan fakta bahwa *tree* yang akan dibuat merupakan pohon biner sehingga untuk setiap *parent node* dengan indeks v maka *left child node* nya memiliki indeks $2v$ dan *right child node* nya memiliki indeks $2v + 1$. Nantinya proses memasukkan nilai pada setiap *node* akan dilakukan secara rekursif dan nilai dari setiap node akan dimasukkan ke *array* yang bersesuaian dengan *node* tersebut. Implementasi fungsinya dalam bahasa C++ adalah sebagai berikut.

```
void build(int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = a[tl];
        return;
    }
    int mid = (tl + tr) / 2;
    build(v * 2, tl, mid);
    build(v * 2 + 1, mid + 1, tr);
    t[v] = t[v * 2] + t[v * 2 + 1];
}
```

Gambar 7. Fungsi Untuk Pembentukan tree
Diambil dari dokumen pribadi

Prosedur di atas memiliki basis ketika nilai $tl = tr$ kemudian pada saat tersebut nilai pada *array* akan dipindahkan ke *tree*. Untuk selain basis, pemrosesan dipisah untuk *tree* bagian kiri dan kanan dan hasil penjumlahan keduanya akan dimasukkan sebagai nilai dari *parents node*. Terlihat bahwa kompleksitas algoritma pembentukan *tree* dapat dihitung dari banyaknya *node* pada *tree* yang nilainya berbanding lurus secara linier dengan jumlah elemen array awal. Oleh karena itu dapat disimpulkan bahwa kompleksitas waktu dari operasi pembentukan *tree* adalah O(n)

2. Pengubahan nilai pada *array*

Pengubahan nilai *array* pada *tree* hanya akan mempengaruhi nilai dari beberapa *node* saja. *Node* yang akan terpengaruh hanyalah *node* yang mengandung *range* dari indeks array yang diubah nilainya. Oleh karena itu pada operasi ini akan dicari *node-node* yang mengandung indeks dari *array* yang nilainya perlu diganti kemudian nilai dari *node* yang perlu diubah akan dicari secara rekursif. Adapun implementasi dari fungsinya dalam bahasa C++ adalah sebagai berikut.

```
void update(int v, int tl, int tr, int pos, int val) {
    if (tl == tr && tl == pos) {
        t[v] = val;
        a[pos] = val;
        return;
    }
    int mid = (tl + tr) / 2;
    if (pos <= mid)
        update(v * 2, tl, mid, pos, val);
    else
        update(v * 2 + 1, mid + 1, tr, pos, val);
    t[v] = t[v * 2] + t[v * 2 + 1];
}
```

Gambar 7. Fungsi Untuk Pengubahan Nilai pada Array
Diambil dari dokumen pribadi

Algoritma di atas memiliki basis yaitu ketika telah ditemukan *node* yang berisi $a[pos]$ kemudian nilainya akan diubah sesuai masukan. Perlu diperhatikan bahwa *node-node* yang nilainya akan berubah hanyalah *node* yang dapat mencapai basis atau dengan kata lain yaitu node yang mengandung informasi dari nilai $a[pos]$. Oleh karena itu hanya *node* yang memiliki keturunan *node* yang mengandung $a[pos]$ yang nilainya perlu diubah. Jumlah dari *node* tersebut tidak lain adalah jumlah *node* antara *node* $a[pos]$ dan *root node*. Jumlah *node* tersebut dapat dihitung dengan mengasumsikan *tree* merupakan sebuah *balanced tree* sehingga level dari *leaf node* adalah $\log_2 n$. Oleh karena itu dapat disimpulkan bahwa kompleksitas waktu dari operasi pengubahan nilai pada *array* adalah O($\log n$).

3. Penjawaban *query*

Proses penjawaban *query* mirip seperti dengan proses pengubahan nilai pada *array* yaitu dengan mencari *node* yang bersesuaian dengan *range query* yang diinginkan. Operasi ini dilakukan dengan pendekatan rekursif dan akan dikembalikan total dari setiap *node* yang sesuai dengan *range* yang dicari. Adapun implementasi fungsinya dalam bahasa C++ adalah sebagai berikut.

```
int get(int v, int tl, int tr, int l, int r) {
    if (tr < l || tl > r) return 0;
    if (l <= tl && tr <= r) return t[v];
    int mid = (tl + tr) / 2;
    int p1 = get(v * 2, tl, mid, l, r);
    int p2 = get(v * 2 + 1, mid + 1, tr, l, r);
    return p1+p2;
}
```

Gambar 9. Fungsi Untuk Penjawaban query
Diambil dari dokumen pribadi

Terdapat dua jenis basis dari fungsi tersebut yaitu ketika *node* yang bersesuaian berhasil ditemukan dan gagal ditemukan. Jika *node* berhasil ditemukan atau dengan kata lain *node* tersebut berada dalam *range* yang sedang dicari maka akan dikembalikan nilai dari *node* tersebut. Sedangkan ketika *node* gagal ditemukan maka akan dikembalikan 0 agar tidak memengaruhi hasil akhir fungsi. Adapun untuk proses rekurensnya sendiri bekerja dengan memecah-mecah *range* ke dua bagian kemudian akan dipanggil fungsi dari kedua *range* tersebut dan hasil keduanya dijumlahkan. Jika dianalisis, kasus terburuk dalam operasi penjawaban *query* yakni ketika *node* yang dicari merupakan *leaf node*. Artinya jumlah operasi yang dibutuhkan adalah jarak dari *root node* ke *node* yaitu $\log_2 n$ sama halnya seperti operasi pengubahan nilai *array*. Oleh karena itu, diperoleh kompleksitas dari operasi penjawaban *query* sebesar $O(\log n)$.

B. Fenwick Tree

1. Penambahan nilai pada *array*

Untuk melakukan penambahan nilai pada posisi *array* tertentu lain halnya dengan *Segment Tree*, pada *Fenwick Tree* prosesnya akan sedikit lebih rumit karena kita perlu memanfaatkan representasi bit dari indeks yang ingin ditambah nilainya. Cara kerjanya yaitu dengan menambah nilai tersebut pada indeks *array* yang dipilih dengan hasil penambahan bit 1 terakhir dari indeks tersebut. Selanjutnya proses akan terus diulangi hingga nilai indeks melebihi panjang *array*. Misalnya kita mempunyai *array* dengan 14 elemen, jika indeks ke 5(0101) yang ingin ditambah nilainya diperoleh bahwa bit 1 terakhirnya adalah 0001 sehingga indeks lain yang perlu diubah terdapat pada posisi $0101 + 0001 = 0110(6)$. Selanjutnya karena indeks bit 1 terakhir dari 6 adalah 0010 maka indeks yang perlu diubah selanjutnya adalah $0110 + 0010 = 1000(8)$. Setelah itu proses akan berhenti dikarenakan $1000 + 1000 = 10000(16)$, sehingga nilai indeksnya telah melebihi panjang *array*. Adapun implementasi fungsinya dalam bahasa C++ adalah sebagai berikut.

```
void add(int idx,int val){
    for(;idx<=n;idx+=idx&-idx){
        f[idx]+=val;
    }
}
```

Gambar 10. Fungsi Untuk Penambahan Nilai pada Array
Diambil dari dokumen pribadi

Pada fungsi tersebut dimanfaatkan operasi bitwise $idx\& - idx$ untuk mendapatkan bit 1 terakhir dari idx. Selanjutnya proses akan terus dilanjutkan hingga idx melebihi panjang array. Pada operasi ini terlihat bahwa operasi yang dijalankan tidak akan melebihi panjang bit dari nilai jumlah elemen *array* sehingga jumlah operasi pada *worst case* adalah $\log_2 n$. Oleh karena itu dapat disimpulkan bahwa operasi penambahan nilai pada *array* memiliki kompleksitas $O(\log n)$.

2. Pembentukan *tree*

Pembentukan *tree* akan memanfatkan fungsi *add* yang telah dibuat sebelumnya. Cara kerjanya yaitu dengan menetapkan nilai *array* awal pada setiap indeks sebesar 0 dan melakukan proses traversal untuk menambah nilai elemen pada setiap indeks *array* ke *tree*. Adapun impelementasi fungsinya dalam bahasa C++ adalah sebagai berikut.

```
void build(int n){
    for(int i=1;i<=n;i++){
        add(i,a[i]);
    }
}
```

Gambar 11. Fungsi Untuk Pembentukan tree
Diambil dari dokumen pribadi

Karena fungsi *add* dipangil sebanyak n kali maka kompleksitas dari operasi pembentukan *tree* adalah $O(n \log n)$.

3. Penjawaban *query*

Operasi penjawaban *query* menggunakan prinsip yang mirip dengan operasi penambahan nilai pada *array* yaitu dengan memanfaatkan bit 1 terakhir. Akan tetapi, pada operasi ini bit 1 terakhir akan menjadi pengurang dari indeks sampai indeks bernilai nol. Misalnya kita mempunyai *array* dengan 14 elemen, jika $idx = 13(1101)$ diperoleh bit 1 terakhirnya yaitu 0001 sehingga indeks selanjutnya adalah $1101 - 0001 = 1100$. Untuk memperoleh indeks selanjutnya diketahui bit 1 terakhir adalah 0100 sehingga indeks selanjutnya adalah $1100 - 0100 = 1000$. Kemudian perhitungan akan berhenti karena $1000 - 1000 = 0$. Jika seluruh indeks yang ditemukan tersebut dijumlahkan akan diperoleh nilai total dari elemen pertama hingga idx. Sehingga untuk menghitung jumlah elemen pada *range* indeks tertentu kita bisa memanfaatkan formula $pref(r) - pref(l-1)$ untuk menghitung total dari *range* $[l..r]$. Adapun implementasi fungsinya dalam bahasa C++ adalah sebagai berikut.

```
int sum(int idx){
    int sum=0;
    for(;idx>0;idx-=idx&-idx){
        sum+=f[idx];
    }
    return sum;
}

int sum(int l,int r){
    return sum(r)-sum(l-1);
}
```

Gambar 11. Fungsi Untuk Penjawaban query
Diambil dari dokumen pribadi

Pada gambar di atas fungsi pertama digunakan untuk menghitung total dari *range* $[1..idx]$, sedangkan fungsi di bawahnya digunakan untuk meghitung total dari *range* $[l..r]$. Mirip halnya pada operasi penambahan, jumlah operasi pada fungsi *sum* bergantung pada panjang bit dari idx. Karena untuk menghitung total pada *range* $[l..r]$ diperlukan pemanggilan fungsi *sum* yang pertama sebanyak 2 kali maka total operasi untuk perhitungan *range* adalah $2\log_2 n$. Oleh karena itu dapat disimpulkan bahwa operasi penjawaban *query* memiliki kompleksitas $O(\log n)$.

V. KESIMPULAN

Dalam menyelesaikan persoalan RSQ, selain menggunakan metode *brute force* dapat pula digunakan struktur data *tree* berupa *Segment Tree* ataupun *Fenwick Tree*. Kedua struktur data tersebut dapat menyelesaikan persoalan RSQ dengan kompleksitas waktu yang efisien. Untuk *Segment Tree*, setiap *query* berupa pengubahan nilai pada *array* serta penjawaban *query* masing-masing bisa dilakukan hanya dengan kompleksitas waktu $O(\log n)$. Sama halnya dengan *Fenwick Tree*, setiap *query* berupa penambahan nilai pada *array* serta penjawaban *query* juga hanya membutuhkan kompleksitas waktu masing-masing yaitu $O(\log n)$. Adapun untuk pembentukan *Segment Tree* membutuhkan kompleksitas waktu $O(n)$ sedangkan pembentukan *Fenwick Tree* membutuhkan kompleksitas waktu sebesar $O(n \log n)$.

Oleh karena itu, penggunaan struktur *Segment Tree* maupun *Fenwick Tree* akan jauh lebih efektif dibandingkan dengan metode *brute force* yang akan menghitung total elemen *array* dengan mencacah indeks satu per satu sehingga kompleksitasnya adalah $O(n)$. Dengan menganggap waktu eksekusi sebuah operasi primitif pada komputer memerlukan waktu 10^{-9} detik maka penggunaan struktur data *tree* tersebut memungkinkan pemrosesan *array* dengan jumlah elemen dan *query* berorde jutaan dalam waktu kurang dari satu detik sehingga dapat menghemat sumber daya komputasi yang ada.

VI. UCAPAN TERIMA KASIH

Puji syukur kehadirat Tuhan yang Maha Esa atas segala rahmat, karunia, serta taufik dan hidayah-Nya sehingga penulis dapat menyelesaikan makalah yang berjudul "Analisis Kompleksitas Operasi pada *Segment Tree* dan *Fenwick Tree* dalam Menyelesaikan Persoalan *Range Sum Query*" sebagai pemenuhan tugas akhir semester I pada mata kuliah Matematika Diskrit IF2120.

Penulis juga ingin berterimakasih terhadap Bapak Dr. Ir. Rinaldi Munir, MT., selaku dosen mata kuliah Matematika Diskrit IF2120 Kelas I yang telah membagikan ilmunya kepada kami, para mahasiswa, selama satu semester ini. Penulis juga beterimakasih kepada seluruh pihak yang telah berkontribusi baik secara langsung maupun tidak langsung terhadap kelancaran penulisan makalah ini yang namanya tidak bisa saya sebutkan satu persatu. Terakhir, penulis ingin mengucapan permohonan maaf apabila terdapat kesalahan dalam penulisan makalah ini.

VII. REFERENSI

[1] https://cp-algorithms.com/data_structures/segment_tree.html Diakses pada 5 Desember 2021

[2] https://www.baeldung.com/cs/segment-trees Diakses pada 5 Desember 2021

[3] https://creatormedia.my.id/cara-kerja-algoritma-brute-force/ Diakses pada 5 Desember 2021

[4] https://cp-algorithms.com/data_structures/fenwick.html Diakses pada 6 Desember 2021

[5] *https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf* Diakses pada 5 Desember 2021

[6] *https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag2.pdf* Diakses pada 5 Desember 2021

[7] *https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf* Diakses pada 5 Desember 2021

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.

Depok, 14 Desember 2021

Primanda Adyatma Hafiz (13520022)