Analisis Kompleksitas Algoritma pada Metode *Square Root Decomposition* dan *Sparse Table* dalam Penyelesaian *Static Range Minimum Query*

Farizki Kurniawan - 13521082[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13521082@std.stei.itb.ac.id

***Abstrak*—*Static Range Minimum Query* adalah persoalan klasik dalam bidang algoritma dan struktur data yang mempertanyakan nilai minimum pada array pada *range* tertentu. Poin penting dalam permasalahan SRMQ ini umumnya adalah jumlah *query* yang cenderung banyak, sehingga memerlukan algoritma pemrosesan query yang cukup efisien untuk dilakukan berulang kali. Beberapa metode untuk menyelesaikan persoalan SRMQ ini ialah *bruteforce, square root decomposition, sparse table, segment tree.* Pada makalah ini, akan ditinjau besaran kompleksitas dari metode *square root decomposition* dan *segment tree* dalam menjawab persoalan SRMQ.**

***Kata Kunci*—Kompleksitas, Sparse Table, Square Root Decomposition, RMQ**

I. Pendahuluan

Range Minimum Query (RMQ) adalah suatu persoalan untuk menjawab beberapa *query*, dengan diberi suatu array A yang berisi *n* buah elemen. *Query* yang didapat berupa dua buah indeks yang terdapat pada array A, *l* dan *r*, dimana terpenuhi syarat $l \leq r$ dan $r \leq n$, dan diperlukan oleh program untuk menjawab nilai elemen minimum di array A pada range *l*-*r* tersebut.

Prefiks *Static* pada *Range Minimum Query* menandakan bahwa pada persoalan ini, tidak akan diberikan *query* tambahan untuk mengubah elemen pada array. Sehingga, dapat dikatakan bahwa array yang dipertanyakan pada setiap *query* adalah sama dan setiap *query* yang identik akan menghasil jawaban yang identik pula.

Persoalan *Range Minimum Query* merupakan persoalan yang umum ditemukan dalam dunia pemrograman. Selain itu, algoritma yang menjawab persoalan *Range Minimum Query* juga secara umum dapat menjawab persoalan sebaliknya, yaitu *Range Maximum* Query, sehingga menyelesaikan persoalan yang satu dapat menyelesaikan persoalan yang lain.

Terdapat banyak alternatif dalam menyelesaikan permasalahan ini, seperti dengan algoritma *bruteforce*, *segment tree*, *square root decomposition*, *sparse table*, dan berbagai algoritma lain. Sehingga, menentukan kompleksitas dari setiap metode yang ada, baik itu dalam kompleksitas waktu ataupun kompleksitas memori menjadi hal yang penting untuk diketahui bagi setiap programmer yang ingin mengimplemen solusi dari permasalahan *Range Minimum Query* tersebut.

Permasalahan RMQ tentunya dapat diselesaikan dengan algoritma yang mudah. Salah satu *naïve approach* dari permasalahan ini ialah dengan menggunakan *bruteforce*. Namun, seringkali pendekatan ini tidak cukup efisien dalam memproses jumlah *query* yang banyak. Untuk *query* yang berjumlah banyak, diperlukan algoritma yang efisien dari memproses setiap *query*. Sehingga, diperlukan informasi yang lebih mengenai berbagai jenis efisiensi dari metode yang bisa digunakan untuk menyelesaikan permasalahan *Range Minimum Query* ini. Untuk itu, dalam makalah ini, akan dibahas tingkat efisiensi dari dua metode lain dalam menyelesaikan masalah RMQ ini, yaitu metode *square root decomposition* dan metode *sparse table.*

II. Landasan Teori

A. Kompleksitas Algoritma

Seringkali, dalam menyelesaikan suatu persoalan pemrograman, tidak hanya diperlukan suatu algoritma yang dapat menemukan jawaban yang tepat, tetapi juga diperlukan algoritma yang dapat melakukannya dengan efisien. Hal ini terlebih penting lagi mengingat beberapa metode pendekatan seperti *bruteforce* dapat memiliki efisiensi yang sangat rendah, sehingga tidak dapat diselesaikan bahkan dalam kurun waktu yang sangat lama ataupun memori yang sangat banyak.

Untuk mengetahui seberapa efisien suatu algoritma akan bekerja, diperlukan suatu perhitungan waktu dan memori. Selain itu, diperlukan juga unit standar dalam penghitungan efisiensi tersebut, seperti jumlah operasi, jumlah alokasi memori, waktu ataupun lainnya, agar kedua jenis efisiensi waktu dan memori dapat terhitung secara kuantitatif. Untuk menghitung kedua jenis efisiensi ini, dapat digunakan kompleksitas algoritma.

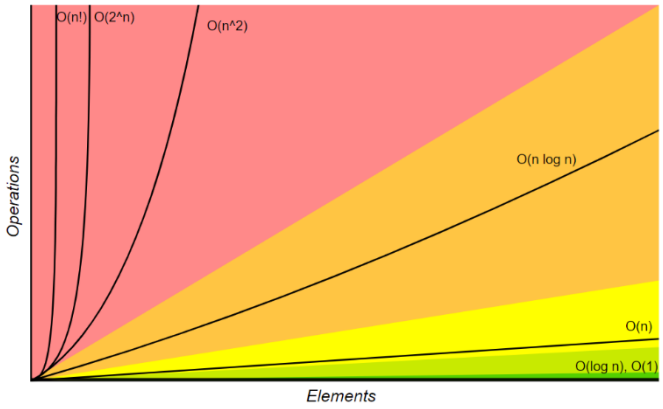
Kompleksitas algoritma adalah suatu besaran yang mengukur efisiensi daripada algoritma tersebut. Kompleksitas operasi sendiri secara umum terbagi menjadi dua, yaitu kompleksitas waktu dan kompleksitas memori. Semakin besar kompleksitas algoritma yang dimiliki suatu metode, maka semakin besar pula waktu ataupun memori, sesuai jenis kompleksitasnya, yang digunakan.

Kompleksitas waktu sendiri diukur dari seberapa banyak operasi dasar yang dilakukan dalam masa jalannya suatu algoritma. Operasi dasar ini dapat meliputi operasi *input*, operasi

output, operasi *assignment*, operasi aritmatika, operasi perbandingan, operasi pengalokasian memori, dan lain sebagainya. Semakin rendah kompleksitas waktu suatu algoritma, maka semakin sedikit total operasi yang dilakukan oleh algoritma tersebut, dan sebaliknya semakin tinggi kompleksitasnya, maka operasi yang dilakukan akan semakin banyak pula.

Kompleksitas memori diukur dari seberapa banyak memori yang digunakan selama algoritma berjalan. Semakin rendah kompleksitas memori suatu algoritma, maka jumlah memori yang diperlukan untuk keberjalanan algoritma akan semakin rendah, dan sebaliknya semakin tinggi kompleksitas memorinya, maka jumlah memori yang diperlukan pun akan semakin tinggi.

Terdapat banyak notasi yang dapat digunakan dalam menentukan kompleksitas algoritma. Salah satu notasi dalam menentukan kompleksitas algoritma yaitu notasi Big-O. Notasi Big-O melambangkan batas atas dari jumlah operasi primitif yang dilakukan oleh suatu program. Suatu algoritma yang memiliki kompleksitas waktu sebesar $O(f(N))$ dapat berjalan dalam waktu berorde paling besar $f(N)$, dengan N adalah besarnya masukan. Perbandingan antara berbagai kompleksitas big-O ialah sebagai berikut

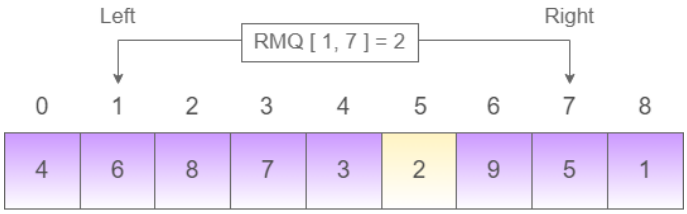


Gambar 2.1 Grafik perbandingan pertumbuhan jumlah operasi beberapa jenis kompleksitas big-Oh
Sumber : bigocheatsheet.com

Secara umum, algoritma yang memiliki kompleksitas waktu dengan orde yang lebih kecil ialah algoritma yang lebih efisien. Algoritma yang paling efisien akan memiliki kompleksitas waktu asimptotik $O(1)$ yang berarti lama jalan algoritma tidak dipengaruhi besarnya masukan.

B. Static Range Minimum Query

Persoalan *Range Minimum Query* adalah persoalan yang bertujuan untuk mengetahui nilai minimum dari setiap elemen yang berada pada sebuah *range* tertentu pada suatu array satu dimensi. Prefiks *Static* pada *Range Minimum Query* menambahkan properti pada persoalan ini dengan menambahkan detail bahwa tidak akan ada perubahan pada array awal yang diberi. Sehingga, metode ataupun data struktur yang memerlukan prekomputasi dengan kompleksitas yang besar namun penjawaban *query* dengan kompleksitas yang minim dapat menjadi opsi yang paling tepat untuk mencapai efisiensi tertinggi.



Gambar 2.2 Ilustrasi Permasalahan Range Minimum Query
Sumber : https://www.algotree.org/images/RMQ.svg

Dalam penyelesaian persoalan *Range Minimum Query*, umumnya ada 2 bagian utama yang perlu diperhatikan. Bagian pertama adalah bagian *preprocessing*, dimana akan dilakukan pemrosesan pada array yang akan ditanyakan untuk menjadi suatu struktur data yang dapat dilakukan *query* dengan lebih efektif. Bagian kedua adalah pemrosesan setiap *query* dengan menggunakan struktur data yang telah diproses.

Langkah *preprocessing* pada *Static Range Minimum Query* menjadi semakin penting karena tidak adanya perubahan dari array yang menjadi array awal pada input. Artinya, terdapat keringanan dalam besarnya efisiensi dari tahap *preprocessing* itu sendiri, karena tahap ini hanya perlu dilakukan sekali selama masa jalannya program.

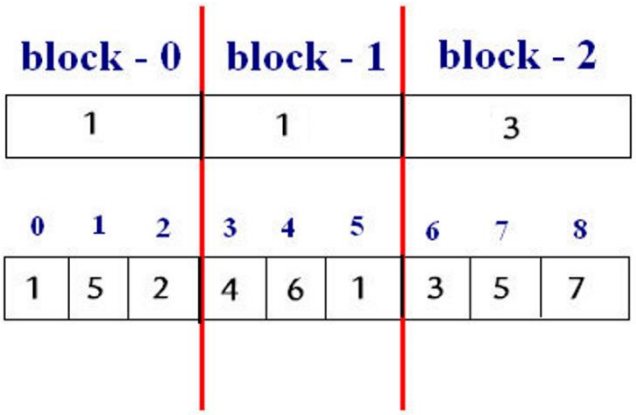
Algoritma pemrosesan *query* yang efektif umumnya menjadi salah satu target penting dalam setiap permasalahan *Range Minimum Query*. Hal ini disebabkan seringkali permasalahan yang timbul adalah terdapat jumlah query yang banyak yang perlu diselesaikan, sehingga jika pemrosesan *query* tidak efektif, program tidak akan dapat memproses seluruh *query* dalam waktu yang dibutuhkan.

C. Square Root Decomposition

Square root decomposition adalah suatu metode yang memecah suatu struktur array menjadi sebanyak $\sqrt{n}$ subarray yang mewakili sebanyak $\sqrt{n}$ elemen dari array awal. Struktur dari setiap subarray ini dapat juga disebut dengan *block*.

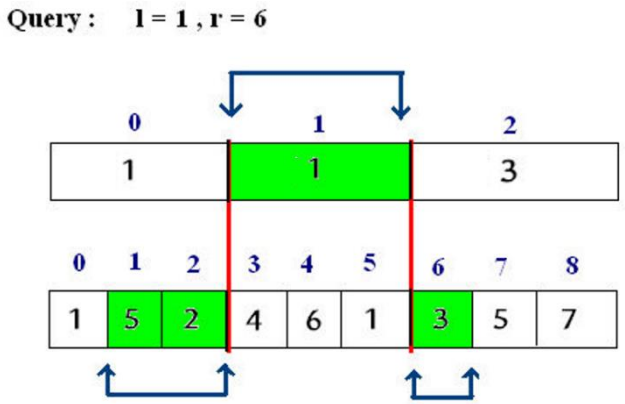
Basis ide dari *square root decomposition* adalah *preprocessing* dari array awal yang dimiliki. Dengan konsep bahwa dengan pemecahan array menjadi bagian-bagian yang dapat dianggap sebagai satu elemen, pemrosesan elemen pada array akan menjadi lebih singkat, kita akan membagi elemen-elemen pada subarray menjadi bagian-bagian yang sama panjang untuk mencapai efisiensi maksimal. Maka dari itu, perlu dilakukan pembagian sebesar $\sqrt{n}$ agar setiap *merged block* yang disatukan dapat memiliki ukuran yang sama sebesar $\sqrt{n}$ pula.

Dalam permasalahan *Range Minimum Query*, setiap struktur *block* menyimpan informasi berupa nilai minimum dari setiap elemen yang terwakilkan oleh *block* tersebut. Dengan menggunakan pembagian per-blok tersebut, kita dapat menghemat jumlah operasi yang dilakukan dengan cara memproses setiap *block* sebagai satu kesatuan, sehingga setiap elemen pada *block* tersebut hanya perlu diproses sebanyak satu kali.



Gambar 2.3 Ilustrasi Square Root Decomposition untuk Permasalahan Range Minimum Query
Sumber : https://ifun01.com/78UHFM5.html

Proses pemrosesan *query* RMQ pada metode *square root decomposition* dilakukan dengan membagi *query* menjadi 3 bagian. Bagian pertamanya ialah semua elemen bagian awal yang tidak terwakilkan oleh *block*. Bagian keduanya ialah semua bagian array yang dapat terwakili secara penuh oleh *block* yang telah dibentuk. Bagian terakhirnya ialah semua elemen bagian akhir yang tidak tertutupi penuh dengan *block* yang ada.

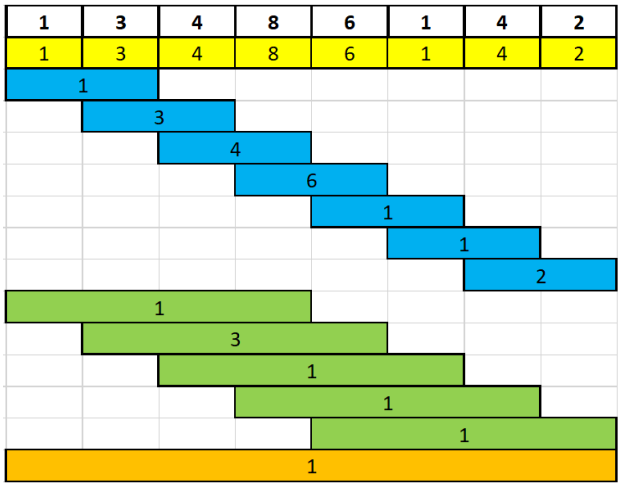


Gambar 2.4 Ilustrasi Pemrosesan Query pada Metode Square Root Decomposition
Sumber : https://ifun01.com/78UHFM5.html

Dari ilustrasi di atas, dapat dilihat bahwa nilai minimum elemen pada indeks 3, 4, dan 5 pada array awal dapat diwakilkan secara langsung oleh block dengan indeks 1. Maka, algoritma hanya perlu membandingkan nilai pada blok tersebut dibandingkan dengan nilai dari elemen array satu per satu. Sedangkan untuk elemen pada indeks 1 dan 2, tidak terdapat blok yang dapat mewakili elemen tersebut secara keseluruhan karena pada *block* dengan indeks 0, terdapat elemen yang juga ikut dibandingkan pada *block* tersebut, yaitu elemen array dengan indeks 0. Maka, perlu dilakukan perbandingan satu per satu dari setiap elemen array tersebut. Hal yang sama juga berlaku dengan elemen pada indeks 6 yang tidak dapat terwakilkan langsung oleh *block* dengan indeks 2.

D. Sparse Table

Sparse table adalah sebuah struktur data yang memanfaatkan properti bahwa setiap bilangan dapat direpresentasikan sebagai pertambahan dari beberapa bilangan pangkat dua yang saling beda. Hal ini secara intuitif dapat kita pahami dengan memahami bahwa setiap bilangan desimal memiliki representasi biner. Lalu, pada representasi biner tersebut, setiap bilangan dapat dianggap sebagai penjumlahan dari dua pangkat jarak ke elemen paling kiri dari setiap angka 1 yang muncul. Sebagai contoh, 14 memiliki representasi biner berupa 1110, atau dapat direpresentasikan sebagai 8 + 4 + 2.



Gambar 2.5 Ilustrasi Sparse Table untuk Permasalahan Range Minimum Query
Sumber : https://materijali.xfer.hr/docs/upiti-nad-intervalima-1/sparse-table/

Dengan properti sebelumnya, dapat kemudian dikatakan bahwa setiap interval juga dapat direpresentasikan sebagai penjumlahan dari beberapa interval berbeda dengan panjang pangkat dua yang saling beda.

Implementasi *sparse table* kemudian dilakukan dengan menyusun sebuah matriks dua dimensi ST, dimana ST[i][j] akan menyimpan jawaban dari *query* untuk *range* j – j + 2^i - 1. Struktur ini mempermudah pengaksesan *range* elemen dengan panjang yang diinginkan.

Pembentukan elemen dari *sparse table* sendiri dapat dilakukan secara sekuensial mulai dari *range* dengan panjang 1, lalu menuju *range* dengan panjang kelipatan dua hingga mencapai nilai *n*, yaitu banyaknya elemen. Nilai dari *range* yang lebih besar dapat dibangun dari 2 *range* lebih kecil yang membentuk *range* tersebut. Dapat dipastikan selalu ada nilai sebelumnya karena algoritma memproses pengisian *sparse table* dari panjang paling kecil. Sebagai contoh, nilai dari *range* [2-5] dapat dibentuk dari nilai pada *range* [2-3] dan *range* [4-5]

Setelah matriks *sparse table* dibentuk, maka pemrosesan *query* dapat dilakukan dengan menggunakan properti dimana setiap bagian *range* ini dapat tertutupi dengan menggunakan dua buah *range* yang paling tidak memiliki panjang sebesar len/2, dengan salah satu *range* dimulai dari dari bagian *l query* dan *range* yang lain diakhiri pada bagian *r query*.

III. Analisa Kompleksitas Penyelesaian Static Range Minimum Query

A. Analisa Kompleksitas Penyelesaian Static RMQ dengan Metode Square Root Decomposition

1. *Preprocessing Array*

Proses *preprocessing array* pada metode *square root decomposition* dilakukan dengan dua buah array, yaitu array yang menyimpan elemen dengan urutan sama seperti elemen input, kemudian array yang menyimpan nilai minimum dari setiap *block* yang berukuran akar dari jumlah elemen.

Prosedur pengisian array kemudian dilakukan dengan pertama-tama mengisi array pertama dengan nilai-nilai input. Selanjutnya, array yang menyimpan *block* akan diisi dengan nilai-nilai minimum pada *range* tersebut di dalam array pertama. Hal ini dilakukan dengan membandingkan semua elemen yang berada pada *block* yang sama dan meng-*assign* nilai tersebut pada indeks *block* yang bersangkutan. Implementasi prosedur ini dalam C++ adalah sebagai berikut.

```
void build(int input[], int n, int A[], int merged[]) {
    // pointer pada array merged
    int p = -1;

    // size sebagai ukuran dari setiap merged block
    int size = ceil(sqrt(n));

    // membangun array A dan merged dari input
    for (int i=0; i<n; i++) {
        A[i] = input[i];
        if (i%size==0) {
            // menuju merged block selanjutaya
            // pointer dimajukan
            p++;
            merged[p]=A[i];
        }

        // isi merged selalu minimum dari semua elemennya
        merged[p]=min(merged[p],A[i]);
    }
}
```

Gambar 3.1 Fungsi Untuk Preprocessing pada Metode Square Root Decomposition
Diambil dari dokumen pribadi

Pada prosedur di atas, terdapat dua parameter input dan dua parameter output. Parameter inputnya berupa array of integers yang merupakan input array pada persoalan RMQ dan juga n yang merupakan jumlah elemen pada array input. Parameter output prosedur ini berupa A, sebagai array of integers yang akan berisi elemen seperti input, dan juga merged, sebagai array of integers yang akan berisi elemen minimum pada setiap block.

Pada algoritma di atas, dapat dilihat bahwa kita hanya mengakses setiap elemen input sebanyak satu kali. Hal ini disebabkan pemrosesan setiap elemen langsung memproses array A dan juga array merged. Maka, dapat dikatakan bahwa kompleksitas waktu dari algoritma di atas ialah sebesar O(n).

2. Pemrosesan *Query*

Proses pemrosesan *query* RMQ pada metode *square root decomposition* dilakukan dengan membagi prosedur *query* menjadi tiga tahap. Tahap pertama dari pemrosesannya ialah untuk memproses semua elemen bagian awal yang tidak terwakilkan oleh *block* pada array merged dengan menggunakan elemen pada array A. Tahap kedua dari pemrosesannya ialah memproses semua bagian array yang dapat terwakili secara penuh oleh *block* yang telah dibentuk dengan menggunakan secara langsung elemen pada array merged. Tahap terakhir dari proses ini ialah memproses semua elemen bagian akhir yang tidak tertutupi penuh dengan *block* yang ada pada array merged. Implementasi prosedur ini dalam C++ adalah sebagai berikut.

```
int query(int merged[], int A[],int l, int r, int n) {
    int mnm=A[l];
    int size = ceil(sqrt(n));
    while (l<r && l%size!=0 && l!=0) {
    // memproses bagian awal yang tidak tercover merged
        if(A[l]<mnm){ mnm=A[l]; }
        l++;
    }
    while (l+size <= r) {
    // memproses bagian A yang tercover merged
        if(merged[l/size]<mnm){ mnm= merged[l/size];}
        l += size;
    }
    while (l<=r) {
    // memproses bagian terakhir yang tidak tercover merged
        if(A[l]<mnm){ mnm=A[l]; }
        l++;
    }
    return mnm;
}
```

Gambar 3.2 Fungsi Untuk Pemrosesan Query RMQ pada Metode Square Root Decomposition

Prosedur *query* di atas memiliki 2 parameter input yang penting, yaitu *l* dan *r*. Dapat dilihat bahwa *l* dianggap sebagai pointer pada fungsi tersebut. While loop yang pertama berfungsi untuk mengiterasi setiap elemen pada bagian pertama yang tidak terwakilkan oleh *merged*, loop yang kedua berfungsi untuk mengiterasi setiap bagian yang dapat terwakilkan oleh merged, dan loop terakhir berfungsi untuk mengiterasi setiap elemen pada bagian terakhir yang tidak terwakilkan oleh *merged*. Jumlah maksimum dari bagian awal yang tidak terwakilkan oleh *block* ialah $\sqrt{n}$, lalu jumlah maksimum dari *block* yang diproses adalah sebesar $\sqrt{n}$, dan jumlah maksimum bagian akhir yang tidak terwakilkan oleh *block* juga sebesar $\sqrt{n}$. Maka didapatkan banyaknya operasi pada algoritma tersebut sebesar $3\sqrt{n}$ dan kompleksitas waktunya sebesar $O(\sqrt{n})$.

3. Kompleksitas Ruang

Dapat dilihat pada implementasi di atas, diperlukan alokasi memori sebesar n untuk array A dan sebesar $\sqrt{n}$ untuk array merged. Maka, dapat kita ketahui bahwa dibutuhkan memori sekitar $n + \sqrt{n}$ untuk menjalankan algoritma tersebut. Sehingga, dengan mengambil suku dengan perkembangan asimtotik paling besar kita dapatkan kompleksitas ruang dari algoritma tersebut ialah O(n).

B. Analisa Kompleksitas Penyelesaian Static RMQ dengan Metode Sparse Table

1. *Preprocessing Sparse Table*

Proses *Preprocessing Sparse Table* dilakukan dengan menggunakan array integer 2 dimensi. Ukuran dari array ini sendiri adalah K x $maxn$, dimana maxn adalah jumlah dari elemen input dan K adalah nilai dari $log_2 maxn$.

Prosedur pengisian *sparse table* kemudian dilakukan dengan pertama-tama mengisi bagian tabel untuk panjang 1, atau sama saja dengan *array* input awal. Kemudian, akan diproses untuk bagian tabel dengan panjang dari kelipatan dua, dimulai dari 2,4,…, hingga 2^K. Pengisian dari tabel ini sendiri dilakukan dengan memanfaatkan fakta bahwa setiap *range* dengan panjang kelipatan 2 sebelumnya telah diproses, sehingga dapat dibandingkan nilai minimum dari kedua bagian yang menyusun *range* yang sedang ditanyakan. Implementasi prosedur ini dalam C++ adalah sebagai berikut.

```
int st[K+1][maxn];
// K >= log2 maxn

void build(int input[], int n){

    // Mengisi Sparse Table dengan panjang 1
    for(int i = 0 ; i < n; i++){
        st[0][i]=input[i];
    }

    // Mengisi Sparse Table dengan panjang 2^i
    for(int i = 1; i < K ; i++){
        for(int j = 0; j + (1<<i) <= n; j++){
            int a = st[i - 1][j];
            int b = st[i - 1][j + (1 << (i - 1))];
            st[i][j] = min(a, b);
        }
    }

}
```

Gambar 3.3 Fungsi Untuk Preprocessing Sparse Table
Diambil dari dokumen pribadi

Prosedur *build* di atas memiliki dua parameter input, yaitu array of integers bernama input, sebagai input array dari permasalahan RMQ, dan n sebagai panjang elemen. Prosedur diawali dengan mengisi sparse table dengan panjang 1 dengan isi dari input array. Hal ini berarti akan terdapat n buah operasi *assignment* pada *sparse table*. Bagian selanjutnya dari prosedur ini ialah pengisian *sparse table* untuk panjang kelipatan 2, dimulai dari 2. Untuk setiap bagian dengan panjang 2^i, terdapat sekitar $n - 2^i + 1$ operasi *assignment* yang perlu dilakukan. Jumlah operasi pada prosedur ini, dengan asumsi n adalah kelipatan 2, ialah sekitar

$$T(n) = (n - 2^0 + 1) + (n - 2^1 + 1) + \cdots + (n - n + 1)$$

atau sama dengan

$$T(n) = n\,log_2\,n - (\,1 + 2 + 2^2 + \cdots + n) + log_2 n$$

dan dapat disederhanakan menjadi

$$T(n) = n\,log_2\,n - (\,2n - 1) + log_2 n$$

Dengan menggunakan suku dengan perkembangan asimtotik paling besar pada T(n), kita dapatkan kompleksitas waktu dari prosedur ini ialah O($n\,log\,n$).

2. Pemrosesan *Query*

Pemrosesan *query* RMQ pada metode *sparse table* dilakukan dengan menghitung terlebih dahulu nilai dari $log_2\,len$, dimana len ialah panjang dari *range* yang ditanyakan. Kemudian, pemrosesan *query* menggunakan properti dimana setiap bagian *range* ini dapat tertutupi dengan menggunakan dua buah *range* yang paling tidak memiliki panjang sebesar len/2, dengan salah satu *range* dimulai dari dari bagian *l query* dan *range* yang lain diakhiri pada bagian *r query*. Dengan membandingkan nilai minimum dari kedua *range* ini yang telah disimpan pada *sparse table* sebelumnya, kita dapat mendapatkan nilai minimum pada *range l-r*. Implementasi dari pemrosesan *query* ini dalam C++ adalah sebagai berikut.

```
int query(int l, int r){

    // Hitung log2 dari panjang range
    int i = log2_floor(r-l+1);

    int mnm = min(st[i][l], st[i][r - ( 1<<i ) +1]);
    return mnm;
}
```

Gambar 3.4 Fungsi Untuk Pemrosesan Query RMQ pada Sparse Table
Diambil dari dokumen pribadi

Pada prosedur di atas, digunakan fungsi log2_floor yang berguna untuk menghitung nilai dari log_2 suatu bilangan. Implementasi fungsinya ialah sebagai berikut.

```
// Fungsi untuk menghitung logaritma i
int log2_floor(unsigned long long i) {
    return i ? __builtin_clzll(1) - __builtin_clzll(i) : -1;
}
```

Gambar 3.5 Fungsi Untuk Mendapatkan Nilai Logaritma
Diambil dari dokumen pribadi

Dari implementasi di atas, dapat dilihat bahwa operasi *assignment* yang dilakukan pada variabel mnm hanya terjadi satu kali, dan operasi pengaksesan elemen *sparse table* hanya terjadi sebanyak 2 kali. Jumlah operasi yang sedikit ini dikarenakan struktur *sparse table* yang memungkinkan prosedur untuk membandingkan 2 nilai saja. Sehingga, dapat dikatakan bahwa kompleksitas waktu dari pemrosesan *query* RMQ pada metode *sparse table* ialah O(1).

3. Kompleksitas Ruang

Dapat dilihat pada implementasi di atas, pada awalnya perlu dilakukan alokasi memori berupa array integer dengan ukuran K x maxn, dimana K adalah nilai yang lebih besar dari dan mendekati dari $log_2\,maxn$ dan maxn adalah jumlah elemen input. Sehingga dapat kita ketahui bahwa dibutuhkan memori sekitar $n\,log\,n$ selama berjalannya algoritma. Sehingga, kita dapatkan kompleksitas ruang dari algoritma tersebut ialah O($n\,log\,n$).

IV. KESIMPULAN

Salah satu metode penyelesaian masalah *Static Range Minimum Query* adalah dengan menggunakan metode *square root decomposition* dan metode *sparse table*. Kedua metode tersebut mempergunakan struktur data yang lebih efisien untuk mempersingkat waktu yang digunakan dalam pemrosesan setiap *query*.

Pada metode *square root decomposition*, kompleksitas waktu dari *preprocessing* array input ialah sebesar O(n), sedangkan pada metode *sparse table*, kompleksitas waktu dari *preprocessing* array input ialah sebesar O($n \log n$). Adapun dalam proses pemrosesan *query*, metode *square root decomposition* memiliki kompleksitas waktu sebesar O($\sqrt{n}$), sementara kompleksitas waktu dari pemrosesan *query* pada metode *sparse table* ialah sebesar O(1). Dalam segi kompleksitas ruang, metode *square root decomposition* memiliki kompleksitas ruang sebesar O(n) dan metode *sparse table* memiliki kompleksitas ruang sebesar O($n \log n$).

Dari kedua metode tersebut, dapat dilihat bahwa keduanya memiliki kompleksitas yang berbeda-beda, serta dapat dilihat bahwa dalam bagian *preprocessing*, metode *square root decomposition* akan lebih cepat dibanding metode *sparse table*, namun dalam pemrosesan *query*, metode *sparse table* akan lebih cepat dibanding metode *square root decomposition.*

V. 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 Algoritma pada Metode Square Root Decomposition dan Sparse Table dalam Penyelesaian Static Range Minimum Query" sebagai pemenuhan tugas makalah pada mata kuliah Matematika Diskrit IF2120.

Penulis juga ingin berterimakasih terhadap Ibu Dr. Nur Ulfa Maulidevi, S.T., M.Sc., selaku dosen mata kuliah Matematika Diskrit IF2120 Kelas K1 yang telah mendidik dan mengajari 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. Penulis juga ingin mengucapan permohonan maaf apabila terdapat kesalahan dalam penulisan makalah ini.

REFERENCES

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian1.pdf Diakses pada 9 Desember 2022
[2] https://www.bigocheatsheet.com/ . Diakses pada 9 Desember 2022
[3] https://cp-algorithms.com/data_structures/sqrt_decomposition.html Diakses pada 7 Desember 2022
[4] https://cp-algorithms.com/data_structures/sparse-table.html Diakses pada 7 Desember 2022
[5] https://materijali.xfer.hr/docs/upiti-nad-intervalima-1/sparse-table/ Diakses pada 9 Desember 2022
[6] https://www.geeksforgeeks.org/sqrt-square-root-decomposition-technique-set-1-introduction/ Diakses pada 8 Desember 2022
[7] Halim, S. Halim, F. Effendy, S. (2020). *Competitive Programming 4.* Noble Education Pte Ltd.

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, 12 Desember 2022

Farizki Kurniawan, 13521082