Segment Tree for Solving Range Minimum Query Problems

Iskandar Setiadi 13511073
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
iskandarsetiadi@students.itb.ac.id

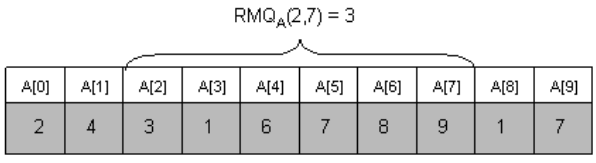
Nowadays, a lot of data structures have been designed to solve problems which are faced in our reality. Complexity is considered as utmost priority in implementing data structure. In this paper, we will discuss a classical problem which is usually noted as Range Minimum Query (RMQ). In this problem, we want to find a minimum value between two specified indices in well-ordered set of array. There are a lot of techniques which can be used to solve this problem and all of them are having varied complexity. One of the most common way in implementing data structures is by using tree. An application of tree, called as segment tree, can be used as an effective approach to solve RMQ. Furthermore, we will compare the complexity of different approaches in solving Range Minimum Query.

***Index Terms*: complexity, data structure, range minimum query, segment tree**

I. INTRODUCTION

Range Minimum Query Problems, denoted as RMQ, is defined as:

> Let A[0..n-1] be a linear array data structure containing n elements of well-ordered set.
> Let i,j be a positive integer such that $0 \le i \le j \le (n-1)$. Let x be a positive integer. For each segment A[i..j], we want to find x ($i \le x \le j$) such that A[x] is position of the most minimum element in sub-array A[i..j].
> A query is denoted as a pair (i,j).



Picture 1.1 Range Minimum Query Representation [7]

Picture 1.1 is a representation of array A containing 10 elements. We want to search position of the most minimum element between i = 2 and j = 7. A query (2,7) is applied at array A and resulting in RMQ_A (2,7) = x = 3. It can be easily seen that A[3] contains the most minimum element between A[2..7], that's it, A[3] = 1.

We can also substitute the stated problem above with maximum one; yet, range minimum query has more applications in our life. This fundamental problem has several applications in database searching, string processing, text compression, text indexing, and computational biology [6]. In computational biology, Range Minimum Query is used in identifying patterns of RNA. By detecting such patterns, biologist can easily identify similarity between two local regions of RNA's.

Imagine that you're given a task to design an application for storing national exam (*Ujian Nasional* in Indonesia) results. Your boss wants to perform thousand operations for searching minimum scores between two intervals. This kind of problem is related to Range Minimum Query Problems. Further explanation of multiple approaches to solve this problem will be discussed later in Section III.

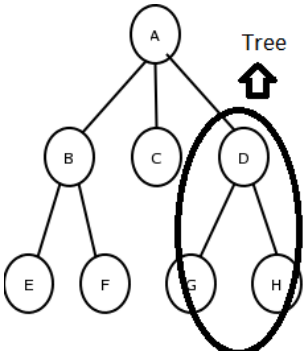
One of the most efficient approach is by using segment tree. There is precisely two steps which are needed in this method. First, we need to build a tree data structure for storing each segments. Lastly, we only need to perform searching by query.

There's also several different approaches such as brute force / trivial method, brute force with DP (Dynamic Programming), and sparse table data structure. On the other hand, different approaches will have different performance too. Several basic knowledge that are needed in this paper will be elaborated in Section II. Every algorithm, which are written in this paper, will be written in C language (as used in IF2030).

II. RELATED THEORIES

A. Tree

Tree is defined as undirected graph and doesn't have any circular vertex [3]. It is classified as a special kind of graph and well-known used to represent hierarchical data structure. Tree data structure also well-known for its recursive properties, as pointed in Picture 2.1 below. Element A is defined as the **parent** of three elements, that's it, element B, C, and D as its **child**.



Picture 2.1 Tree Data Structure

Let G = (V,E) is undirected graph and having n edges.
A tree will have following properties:

- Each vertices in G have a singular path
- G is connected and it has (n – 1) edges
- If an edge is removed, graph will be separated into two different components
- G has no cycles and any two particular vertices can be connected by using one simple path

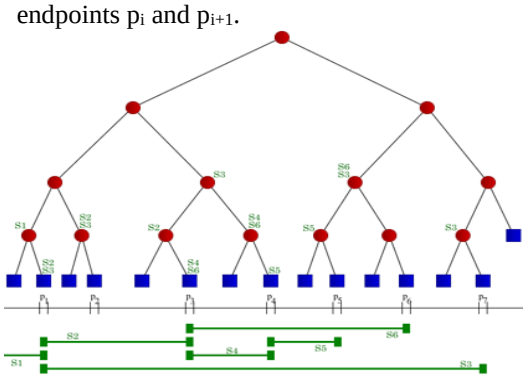
Several basic tree terminologies that will be used in this discussion are listed below:

- **Root node** : the topmost node in the tree (A)
- **Leaf node** : the most bottom node in the tree (E, F, G, H)
- **Height** : the longest length from the topmost node to a leaf node

B. Segment Tree

One sub-application of tree data structure is segment tree. It allows querying which of the stored intervals contain a given value. The definition of segment tree is given below:

> Let p_i be the list of distinct segment indices, defined as $1 \leq p_i \leq n$, for i = 1,2,3,… . A segment tree can be partitioned into several regions which intervals are defined with two consecutive endpoints p_i and p_{i+1}.



Picture 2.2 Segment Tree Data Structure [1]

[1] Image Reference, December 17, 2012 (02.00 AM) <http://upload.wikimedia.org/wikipedia/commons/e/e5/Segment_tree_instance.gif>

Picture 2.2 shows an example of the structure of segment tree. Each nodes in segment tree are storing value between intervals. Hence, segment tree can be classified as interval tree.

C. Complexity

Complexity is used as an efficiency parameter in computational programming. Big-O notation is usually used to describe the behaviour of particular function, which is simplified in simpler terms. Let f, T be a positive non-decreasing function defined on positive integer. We can infer that T(N) = O(f(N)) if for some constant C and n_0 holds:

$$\forall N > N_0;\ T(N) \leq O(f(N))$$

As stated above, we usually simplify this behaviour into simpler terms. Table 2.1 below is the classification of the following notation:

Complexity (Notation)	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
O(n!)	Factorial

Table 2.1 Order of Time Complexity in Big-O Notation

> **Example II.C – 1** You're given the following algorithm, written in C language. Determine its classification (using big-O notation).
>
> ```
> for(i=0;i<n;i++)
>   for(j=0;j<n;j++)
>     A[i,j] = 0;
> ```

Running through j = 0 to j = n – 1 requires T(n) = n while running through i = 0 to i = n – 1 also requires T(n) = n. Hence, Example II.C – 1 can be classified as T(n) = O(n * n) = $O(n^2)$, which is running in quadratic time.

A boundary of limited runtime and memory space is usually found in competitive programming. Not limited to such particular area, every programmers want to design faster algorithms which are using less memory space. An enormous differences occur when we're comparing several optimized algorithms with un-optimized ones.

> **Example II.C – 2** Suppose that you're given one million elements within linear array data structure. You want to compute one million queries in searching unique element indices.

In order to solve Example II.C-2, we'll use two different approaches. Traverse-search requires $T_{max}(N) = 10^6 * 10^6 = 10^{12}$ while binary-search requires $T_{max}(N) = 10^6 * {}^2\log(10^6) \approx 2.10^7$. Traverse-search is running in linear time T(n) = O(n) while binary-search is running in logarithmic time T(n) = O(log n). It can be easily seen that binary-search performs faster than traverse-search.

D. Dynamic Programming

One of the most infamous technique in computer science is dynamic programming. It's a method which is usually based on recurrence; hence, a problem can be break down into simpler sub-problems. The main principle of dynamic programming is by storing previous computation of a sub-problem, well-known as memoization. For better understanding, let's take a look to Example II.D – 1.

Example II.D – 1 Fibonacci Sequence, f (n) = f(n-1) + f(n-2) for n > 2 ; f(1) = 1, f(2) = 1. Design an algorithm to compute f(10.000) !

Using trivial / brute force technique, f(10.000) will call the function f(9.999) and f(9.998). Repeatedly, f(9.999) will also call the function f(9.998), which is classified as redundant operation. This kind of technique leads into an exponential time $O(2^n)$.

Using dynamic programming, it is possible to design such algorithm that running in linear time O(n) instead of exponential time $O(2^n)$. Let A[1..10.000] be a linear array containing 10.001 elements of positive integer. For each calculation, we'll define:

$$A[n] = A[n-1] + A[n-2]; n > 2, A[1] = 1, A[2] = 1$$

In this computation, we will use bottom-up approach. First, the function A[3] will call the function A[2] and A[1]. At the next step, A[4] will call the function A[3] and A[2], which is previously stored in A[3] and A[2]. This recursive-stored method only requires O(n) memory space and O(n) running time. In terms of complexity, one can infer that DP method is far more effective against brute force technique.

Both segment tree data structure and sparse table data structure are related to dynamic programming. These technique compute RMQ at interval [i, j] by using two previous intervals ; $i \neq j$. The following computation can be written as:

$$[i, j] = \min\left(\left[i, i + \left\lfloor \frac{j-i}{2} \right\rfloor\right], \left[i + \left\lfloor \frac{j-i}{2} \right\rfloor + 1, j\right]\right)$$

The main difference between segment tree and sparse table is usage of memory space. Segment tree is using tree data structure, while sparse table is using 2-D array data structure. Otherwise, segment tree is considered more powerful than sparse table because of its flexibility.

E. Sparse Table

Sparse table data structure is created by two-dimensional array, sized of [0..n-1, 0..log(n)] whereas n can be written as 2^k. Let A[0..n-1, 0..log(n)] be a two-dimensional array data structure containing n elements, denoted as A[i, 0] ; $0 \leq i \leq n$. This technique is named as sparse table because its matrix is populated primarily with zeros as elements of the table.

Example II.E – 1 You're given 8 elements, 5, 10, 12, 8, 4, 7, 2, and 10 consecutively. Implement sparse table data structure within these elements!

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	
5	10	12	8	4	7	2	10	0
5	10	8	4	4	2	2	0	1
5	4	4	2	2	0	0	0	2
2	0	0	0	0	0	0	0	3

Picture 2.3 Minimum Value Sparse Table

For better understanding in implementing sparse table, see Example II.E – 1. Let A[0, 0] = 5, A[1, 0] = 10, … , A[7, 0] = 10. For each i, j positive integer, A[i, j] in Picture 2.2 is the minimum value between indices i and $i + (2^j - 1)$. For example, we want to know the minimum value between indices 2 and 6. We only need to compare minimum value between [2..5] and [3..6] (O(1)). It can be written as:

$$\min[2..6] = \min(A[2, 2], A[3, 2]) = \min(4, 2) = 2$$

The minimum value between indices 2 and 6 is 2 (positioned as A[6]).

III. Solving Range Minimum Query

A. Brute force / Trivial Approach

One of the most trivial algorithm to solve range minimum query is by using brute force approach. For each pair of indices (i,j), we only need to store the position of every minimum value in matrices. This algorithm uses $O(n^2)$ memory space and $O(n^3)$ running time complexity.

Let A[0..n-1, 0..n-1] be a matrices of integer. Let B[0..n-1] be a linear array data structure containing n elements of well-ordered set. $RMQ_B(i,j)$ is denoted by A[i, j]. A[i, j] is defined by the position of minimum value between B[i] and B[j]. This computation of query searching requires O(1) running time complexity. Following implementation of brute force approach is written in C language:

```
/* Main Algorithm */
for(i=0;i<n;i++)
 //Initial state
  A[i][i] = i;
 for(i=0;i<n-1;i++)
  for(j=i+1;j<n;j++)
   //Initial state
   temp = i;
   A[i][j] = temp;
   for(k=i+1;k<=j;k++)
     //Searching for RMQ value
       if (B[k] < B[temp])
         {
           A[i][j] = k;
           temp = k;
         }
```

B. Trivial Approach Using Dynamic Programming

As previously explained, implementation with brute force approach will have $O(n^3)$ running time complexity. It's considered ineffective, especially for large cases (n > 500). However, by using a dynamic programming approach, we can reduce the complexity from $O(n^3)$ to $O(n^2)$. Dynamic programming is a method to memoized previous computation, which will be used for next computation processes. This computation of query searching also requires O(1) running time complexity. Following is the implementation of trivial approach (using DP):

```
/* Main Algorithm */
for(i=0;i<n;i++)
 //Initial state
  A[i][i] = i;
for(i=0;i<n-1;i++)
  for(j=i+1;j<n;j++)
   //Searching for RMQ value
   //Dynamic programming approach
   if (B[A[i][j-1]] < B[j])
       A[i][j] = A[i][j-1];
   Else
       A[i][j] = j;
```

C. Sparse Table Data Structure

Another approach to solve range minimum query problem is by implementing sparse table. As explained in Section II, sparse table is created by matrices (two-dimensional array). In this approach, we will combine sparse table data structure with dynamic programming technique. This algorithm uses O(n log n) memory space and O(n log n) running time complexity. For query searching, we only require O(1) running time complexity.

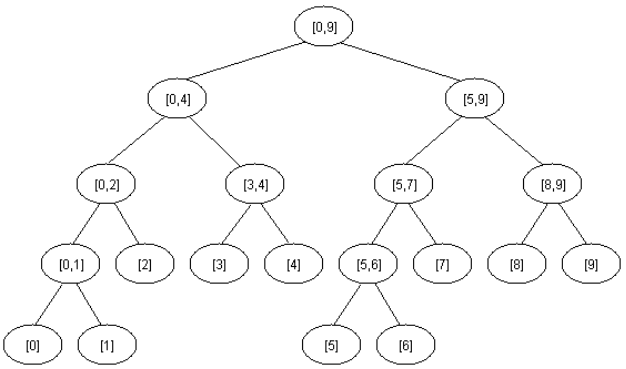
Let A[0..n-1, 0..log(n)] be a two dimensional array where A[i, j] is the position of the minimum value between indices i and $i + (2^j - 1)$. In this manner, we can solve range minimum query problem with a simple comparison between two segments. The implementation of sparse table with C language can be written as:

```
/* Main Algorithm */
for(i=0;i<n;i++)
 //Initial state
  A[i][0] = i;
for(j=1;1<<j<=n;i++)
  for(i=0;i+(1 << j)-1<n;j++)
   //Searching for RMQ value
   //Sparse table data structure
   if (B[A[i][j-1]] < B[A[i+ 1 <<
(j-1)][j-1]])
       A[i][j] = A[i][j-1];
   else
       A[i][j] = A[i+(1 << (j-
1)][j-1];
```

Operator "<<" is defined as bitwise shift operators [11]. It is used to represent 2^j in our algorithm. Further explanation of bitwise operator will not be discussed here.

D. Segment Tree Data Structure

One of the most powerful technique in solving range minimum query problem is by using segment tree data structure. In order to build a segment tree, we only need O(n log n) memory space and O(n log n) running time complexity. This is considered very effective because there isn't any redundant memory space which is used in this technique. For comparison, sparse table requires 256 MB ($O(n^2)$) of memory while segment tree only requires 64 MB (O(n log n)), which is far more effective for large cases.



Picture 3.1 Illustration of Segment Tree

For better understanding, let's take a look at Picture 3.1. In order to construct its binary search tree, we will use a recursive, bottom-up method. For example, the process of interval [0,9] will use the result of interval [0,4] and [5,9].

Let MaxI be a constant sized of $2^{2\log(n) + 2}$, defined as total elements in A[0..MaxI-1] and MaxN be a constant sized of n, defined as total elements in B[0..MaxN-1]. Let all elements in A[0..MaxI-1] are initialized with -1. Following is the implementation of segment tree data structure (using BuildTree function):

```
 void BuildTree (int node, int b,
int e, int A[MaxI], int B[MaxN], int
n)
 {
   /* Main Algorithm */
    if (b == e)
      A[node] = b;
    else
    {
      //Recursive subtrees
      BuildTree (2 * node, b,
(b+e)/2, A, B, n); //Left
      BuildTree (2 * node + 1, b,
(b+e)/2 +1, A, B, n); //Right

      //Search for RMQ value
      if (B[A[2 * node]] <= B[A[2 *
node + 1]])
          A[node] = A[2 * node];
      else
          A[node] = A[2 * node + 1];
    }
 }
```

We've already built the segment tree for our queries searching process. We can now start making queries. Every processes of query searching is done in O(log n) running time, which is practically fast enough. Segment tree can also be modified dynamically, which is practically better than segment tree for large cases. Following is the implementation of query searching function (using QuerySearch function):

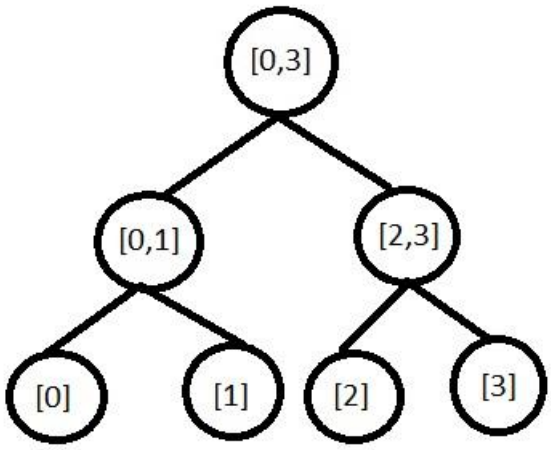
```
 int QuerySearch (int node, int b,
int e, int A[MaxI], int B[MaxN], int
i, int j) //Interval (i,j)
 {
   /* Local Dictionary */
   int p1, p2;
   /* Main Algorithm */
   //If doesn't intersect
   if ( i > e || j < b)
     return -1;

   //If included in current node
   if ( b >= i && e <= j)
     return A[node];

   //Recursive binary tree method
   //Compute from left & right
   p1 = QuerySearch (2 * node, b,
(b+e)/2, A, B, i, j);
   p2 = QuerySearch (2 * node + 1,
b, (b+e)/2 + 2, A, B, i, j);

   //Return position of indices
   if (p1 == -1)
     return A[node] = p2;
   if (p2 == -1)
     return A[node] = p1;
   if (B[p1] <= B[p2])
     return A[node] = p1;
   return A[node] = p2;

 }
```

IV. Example and Solution

In this section, we will use our newest tool, which is well-known as segment tree, to solve a range minimum query problem.

Example IV – 1 One day, an economic crisis has risen and government has already took several steps to prevent this crisis. Consider Indonesia has M number of families ($1 \le M \le 100.000$). Each families have monthly income, denoted as V_p ($0 \le p \le M-1$). BLT (*Bantuan Langsung Tunai*) is distributed to N families ($1 \le N \le 100.000$) and each package of BLT can only be distributed into exactly one family between (i,j) interval ($0 \le i \le j \le M-1$). Determine all families that will receive these packages of BLT.

For better understanding, we'll take smaller dataset before solving this problem thoroughly. Assume that we have M = 4, and you're given $V_0 = 7$, $V_1 = 1$, $V_2 = 4$, and $V_3 = 3$. Using BuildTree function, the intervals of segment tree are represented in Picture 4.1 below.



Picture 4.1 Segment Tree Intervals for Sample Problem

The value of each interval can be written as:
- [0, 3] = 1
- [0, 1] = 1 and [2, 3] = 3
- [0] = 7, [1] = 1, [2] = 4, and [3] = 3

For example, we want to search the minimum value between (i,j) = (2,3). We'll use the QuerySearch function:

```
QuerySearch(1, 0, M-1, A, B, 2, 3) 4.1
```

At the function calls above, we are possible to derive its generalization function formula as:

```
QuerySearch(1, 0, M-1, A, B, i, j) 4.2
```

The function will give us an integer value between $0 \le i, j \le M-1$ if and only if $i \le j$. Otherwise, the QuerySearch function will return -1.

After finishing our analysis for small dataset, we'll solve this problem thoroughly. Following is the solution of Example IV – 1:
- Declare MaxN as M ($1 \le M \le 100.000$)
- Declare MaxI as $2^{2\log(M)+2}$
- Store each value of V_p in B[0..MaxN-1]
- Initialize all elements in A[0..MaxI] with -1
- Create its Segment Tree using BuildTree
- For each query, we just simply call function 4.2

We've simply finished solving this problem thoroughly. This algorithm works with O(n log n) running time for BuildTree function, O(n log n) memory space, and O(log n) running time for QuerySearch function. Within such limitation $1 \le M \le 100.000$ and $1 \le N \le 100.000$, it requires less than 1 second in newest processor technologies.

V. Conclusion

Tree data structure has widely applications in computer science because of its flexibility. Segment tree, which is classified as interval tree, has several applications in term of computational areas. One of them is to solve range minimum query (RMQ) problem. Range minimum query is used to find the position of minimum value between two local intervals.

There are at least four techniques, which are broadly used in solving RMQ problem (Section III). One of the most important aspect in computational programming is complexity. Complexity is usually determined from the running time and memory space usage of an algorithm. One of the most flexible and efficient method is by using segment tree data structure.

The usage of tree data structure is very broad; a lot areas of science use this kind of representation in order to solve scientific problems.

VI. Acknowledgment

Iskandar Setiadi, as the author of this paper, want to express his deepest gratitude to Dra Harlili, M.Sc and Dr. Ir. Rinaldi Munir, M.T. as the lecturers of IF 2091 – "*Struktur Diskrit*". Special thanks to my family and all my friends of Informatics 2011.

References

[1] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. 2009. *Introduction to Algorithms* 3rd ed. MIT Press and Mc Graw-Hill.

[2] Halim, Steven & Felix Halim. 2010. *Competitive Programming 1st* ed. Lulu Publisher.

[3] Munir, Rinaldi. 2008. *Diktat Kuliah IF 2091 Struktur Diskrit* 4th ed. Program Studi Teknik Informatika STEI ITB.

[4] de Berg, Mark; van Kreveld, Marc; Overmars, Mark; Schwarzkopf, Otfried. 2000. *Computational Geometry: algorithms and application 2nd* ed. Springer-Verlag Publisher.

[5] Rosen, Kenneth H. 2003. *Discrete Mathematics and Its Application 5th* ed. Mc Graw-Hill.

[6] Fischer, Johannes. 2008. *Range Minimum Queries: Simple and Optimal, at Last!* December 16, 2012 (03.00 AM). < *http://www.bio.ifi.lmu.de/~fischer/fischer09range.pdf*>

[7] Topcoder. *Range Minimum Query and Lowest Common Ancestor*. December 15, 2012 (05.00 PM). <http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor>

[8] Topcoder. *Computational Complexity*: Section One. December 15, 2012 (11.00 PM). <http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=complexity1>

[9] NTHU. *Interval Tree and Related Problems.* December 17, 2012 (01.40 AM). < http://www.cs.nthu.edu.tw/~wkhon/ds/ds10/tutorial/tutorial6.pdf>

[10] Fredman, Michael L., Janos Komlos. 1984. *Storing a Sparse Table with O(1)*. December 16, 2012 (01.00 PM). <http://www.cs.princeton.edu/courses/archive/fall09/cos521/Handouts/storingasparse.pdf>

[11] PJ Arends. *An Introduction to Bitwise Operators*. December 17, 2012 (04.50 PM). <http://www.codeproject.com/Articles/2247/An-introduction-to-bitwise-operators>

Statement

I hereby stated that this paper is copyrighted to myself, neither a copy from other's paper nor a translation of similar paper.

Bandung, December 18, 2012

Iskandar Setiadi 13511073