# The Range Minimum Query Problem

Iulia Andreea Naomi Mihaicuta

Politehnica University of Bucharest

**Abstract.** The paper studies the Range Minimum Query Problem (finding the minimal element in a sub-array) and its existent solution. Based on their performance and complexity, the following methods are evaluated: Sqrt Decomposition, Segment Tree and Sparse Table.

**Keywords:** Range Minimum Query Problem · RMQ · Square Root Decomposition · SQRT Decomposition · Segment Tree · Sparse Table · Lowest Common Ancestor Problem · LCA

## 1 Introduction

The Minimum Range Query problem involves finding the smallest element in an array between two given indexes, that is, given the array $\vec{a}$[0, N - 1], with N elements, and the two positions i and j, with i < j, $RMQ_{\vec{a}}(i, j)$ returns the position of the minimum element in $\vec{a}[i, j]$.

$$RMQ_{\vec{a}(i,j)} = argmin_{i \leq k \leq j} \vec{a}_k$$

For example, when $\vec{a} = [4, 6, 3, 23, 45, 15, 23, 2, 14]$, a range minimum query for the sub-array $\vec{a}$[2, 6] returns 3.

There are M queries of the program and between any of them the array can be altered as follows: upon receiving $p_i$, the position of the element that is to be modified, the element is given the new value $v_i$.

$$\vec{a}_{p_i} \leftarrow v_i$$

The RMQ data structure has a time complexity of ⟨p(n), q(n)⟩ where p(n) is the complexity of preprocessing and q(n) is the complexity of a query.

There are several practical appications that can be reduced to the Range Minimum Query Problem, including finding the Lowest Common Ancestor of two nodes. The LCA problem has applications in models of complex systems found in distributed computing as well as in the implementation of object-oriented programming systems. It is also possible to apply the RMQ problem to 3D applications through Occlusion Culling.

### 1.1 Solution Overview

**Square Root Decomposition**

In Square Root Decomposition, the array is divided into buckets of size n, each of which stores the minimal value. The complexity of preprocessing is $O(n)$. By using this approach, the number of operations is significantly reduced, resulting in a time complexity of $O(\sqrt{n})$ per query.

**Segment Tree**

Since the method has a short build time and can update array values rapidly, it is commonly used when array values need to be updated.

Query complexity is $O(log(n))$, while preprocessing complexity is $O(n)$.

**Sparse Table**

Sparse Table represents a solution where the complexity of a query is $O(1)$, but the complexity of preprocessing is $O(nlog(n))$. This is the reason why the concept is mainly used for fast queries on a static data set.

### 1.2 Evaluation Criteria

Algorithms are evaluated using 4 types of test inputs, as follows:

**Tests 01 - 16** - wide range of values for the lengths of the queries.
**Tests 17 - 32** - the length of each query being close to the size of the array.
**Tests 33 - 48** - the length of each query being close 500.
**Tests 49 - 64** - the length of each query being close 5.

## 2 Solutions

### 2.1 Square Root Decomposition

The array $\vec{a}$ is divided into blocks of length approximately $s = [\sqrt{n}]$, and for every block $i$, the minimum element is saved in $\vec{b}[i]$. The array $\vec{a}$ is partitioned into blocks as follows:

$$\underbrace{a[0], a[1], \ldots, a[s-1]}_{\text{b[0]}}, \underbrace{a[s], \ldots, a[2s-1]}_{\text{b[1]}}, \ldots, \underbrace{a[(s-1)\cdot s], \ldots, a[n-1]}_{\text{b[s-1]}}$$

For each block $k$, $ca$ $\vec{b}[k] = min(\vec{a}[ks], \vec{a}[ks+1], \ldots, \vec{a}[t])$, where:

$$t = min(n-1, (k+1) * s - 1)$$

If the interval $[q_s{}^1, q_e{}^2]$ is long enough, it will contain some integral blocks, but also two "tails". In order to find the minimum element, we only need to compare the elements of the two "tails" with the values of $\vec{\mathrm{b}}[i]$; $i \in [k+1, p-1]$.

$$A = min(\vec{\mathrm{a}}[q_s], \vec{\mathrm{a}}[q_s + 1], \ldots, \vec{\mathrm{a}}[(k+1) * s - 1])$$
$$B = min(\vec{\mathrm{b}}[k+1], \vec{\mathrm{b}}[k+2], \ldots, \vec{\mathrm{b}}[p-1])$$
$$C = min(\vec{\mathrm{a}}[p * s], \vec{\mathrm{a}}[p * s + 1], \ldots, \vec{\mathrm{a}}[q_e])$$
$$RMQ_{\vec{\mathrm{a}}(q_s, q_e)} = min(A, B, C)$$

**Advantages and Disadvantages** Square Root Decomposition allows RMQ to be solved in $O(\sqrt{n})$ time, which is faster than other techniques such as segment trees - $O(log(n))$ or the naive approach - $O(n)$, and it uses $O(\sqrt{n})$ space. Also, SQRT Decomposition is relatively simple to implement. However, the technique is not as flexible as other RMQ solution, as it cannot handle updates to the array as efficiently - $O(\sqrt{n})$. Additionally, the method performs poorly when dealing with small arrays or large ranges.

### 2.2 Segment Tree

A segment tree is a data structure that can be used to efficiently solve the Range Minimum Query problem. Each node represents a range of the array, and its value is the minimum element within that range.

Sub-arrays of size 1 (i.e. individual elements) are created from the array. In the segment tree, each of these sub-arrays is represented by a leaf node. The internal nodes represent the merged range of their child nodes. The value of each internal node in the tree is the minimum of the values of its child nodes.

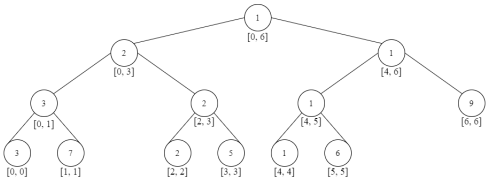


**Fig. 1.** Segment tree corespondent to $\vec{\mathrm{a}} = [3, 7, 2, 5, 1, 6, 9]$

By traversing the tree and comparing values at the nodes, we can efficiently solve RMQ by using the segment tree. Thus, RMQ can be solved in $O(log(n))$ time, where n is the size of the array.

[1] index representing the start of the interval
[2] index representing the end of the interval

**Advantages and Disadvantages** Segment trees allow RMQ to be solved in $O(logn)$ time, which is faster than other techniques such as brute force, which have a time complexity of $O(n)$. The method performs well on both small and large arrays. Unlike some other techniques, the segment trees are capable of efficiently handling updates to the array, such as deleting or inserting elements. However, the storage space complexity of segment trees is higher, as they require $O(n)$ space. They are also more complex to implement and have a poor performance on ranges that are much smaller than the array - $O(log(n))$ is still required.

### 2.3 Sparse Table

Sparse Tables ease querying of the minimum(or maximum) element in an array. They are especially useful when arrays are static and queries are frequent. Creating a sparse table requires preprocessing the array by constructing o two-dimensional array, where each element represents the minimum element in the sub-array.

Unlike other data structures, Sparse Table is limited to immutable arrays. If any element changes in between two queries, the whole structure must be recalculated.

The first column of the table contains the elements of the original array. Considering k the column, for each row in the column , compute the minimum element in the sub-array starting at index i and of length $2^k$ by comparing the minimum element in the sub-array starting at index i and of length $2^{k-1}$ with the minimum element in the sub-array [3] starting at index $i + 2^{k-1}$ and of length $2^{k-1}$.

**Advantages and Disadvantages** Sparse Tables have the advantage of answering queries in constant time, making them faster than most methods for resolving the RMQ problem. Comparatively to other methods that require linear space, Sparse Table has a space complexity of $O(nlog(n))$. Also, it can handle large input sizes efficiently. However, some of the disadvantages of using this data structure include an high preprocessing time - it requires $O(nlog(n))$ preprocessing time, whereas Segment Tree only requires $O(n)$ - and not being suitable for updates - due to the need to reprocess the entire array every time the input array changes, Sparse Table is not suitable for dynamic situations when the input array is updated frequently.

[3] Sparse Table - Algorithms for Competitive Programming

# 3 Evaluation

## 3.1 Results and Explanations

The tests were run on a AMD Ryzen 7 4800H CPU @ 2.90GHz, 16GB of RAM with Windows 11 Home, 64-bit operating system. The algorithms tested are Segment Tree, Sparse Table and SQR Decomposition.

| Test Index | N | M | SQR Decomposition | Segment Tree | Sparse Table | Best Solution |
|---|---|---|---|---|---|---|
| 1 | $10^3$ | $10^3$ | 1218.33 | 0.00 | 0.00 | Segment Tree |
| 2 | $10^3$ | $10^4$ | 1188.03 | 0.00 | 0.00 | Sparse Table |
| 3 | $10^3$ | $10^5$ | 1203.39 | 75.26 | 31.27 | Sparse Table |
| 4 | $10^3$ | $10^6$ | 1187.77 | 498.33 | 296.85 | Sparse Table |
| 5 | $10^4$ | $10^3$ | 1203.38 | 2.99 | 0.00 | Sparse Table |
| 6 | $10^4$ | $10^4$ | 1187.77 | 10.00 | 15.63 | Segment Tree |
| 7 | $10^4$ | $10^5$ | 1188.26 | 58.98 | 31.24 | Sparse Table |
| 8 | $10^4$ | $10^6$ | 1203.12 | 565.69 | 312.68 | Sparse Table |
| 9 | $10^5$ | $10^3$ | 1203.30 | 13.00 | 62.51 | Segment Tree |
| 10 | $10^5$ | $10^4$ | 1188.00 | 17.99 | 62.49 | Segment Tree |
| 11 | $10^5$ | $10^5$ | 1188.21 | 74.96 | 93.74 | Segment Tree |
| 12 | $10^5$ | $10^6$ | 1172.68 | 631.72 | 390.62 | Sparse Table |
| 13 | $10^6$ | $10^3$ | 1187.61 | 109.37 | 718.87 | Segment Tree |
| 14 | $10^6$ | $10^4$ | 1187.70 | 124.97 | 703.12 | Segment Tree |
| 15 | $10^6$ | $10^5$ | 1172.88 | 187.95 | 734.77 | Segment Tree |
| 16 | $10^6$ | $10^6$ | 1187.50 | 750.55 | 1047.41 | Segment Tree |

**Table 1.** Algorithm's performance on tests with a wide rage of query lenghts.

| Test Index | N | M | SQR Decomposition | Segment Tree | Sparse Table | Best Solution |
|---|---|---|---|---|---|---|
| 17 | $10^3$ | $10^3$ | 1188.00 | 0.00 | 0.00 | Segment Tree |
| 18 | $10^3$ | $10^4$ | 1187.76 | 0.00 | 0.00 | Segment Tree |
| 19 | $10^3$ | $10^5$ | 1187.95 | 46.87 | 31.23 | Sparse Table |
| 20 | $10^3$ | $10^6$ | 1187.83 | 422.08 | 265.63 | Sparse Table |
| 21 | $10^4$ | $10^3$ | 1187.91 | 0.00 | 0.00 | Segment Tree |
| 22 | $10^4$ | $10^4$ | 1187.98 | 15.62 | 15.62 | Segment Tree |
| 23 | $10^4$ | $10^5$ | 1203.85 | 47.06 | 31.24 | Sparse Table |
| 24 | $10^4$ | $10^6$ | 1187.54 | 485.29 | 281.46 | Sparse Table |
| 25 | $10^5$ | $10^3$ | 1188.01 | 15.62 | 62.57 | Segment Tree |
| 26 | $10^5$ | $10^4$ | 1188.32 | 15.63 | 62.48 | Segment Tree |
| 27 | $10^5$ | $10^5$ | 1204.07 | 78.11 | 78.10 | Sparse Table |
| 28 | $10^5$ | $10^6$ | 1193.60 | 578.13 | 343.88 | Sparse Table |
| 29 | $10^6$ | $10^3$ | 1182.52 | 109.36 | 703.15 | Segment Tree |
| 30 | $10^6$ | $10^4$ | 1203.36 | 125.37 | 703.36 | Segment Tree |
| 31 | $10^6$ | $10^5$ | 1203.54 | 187.48 | 734.34 | Segment Tree |
| 32 | $10^6$ | $10^6$ | 1187.50 | 750.36 | 984.64 | Segment Tree |

**Table 2.** Algorithm's performance on tests with the length of each query being close to the size of the array.

| Test Index | N | M | SQR Decomposition | Segment Tree | Sparse Table | Best Solution |
|---|---|---|---|---|---|---|
| 33 | $10^3$ | $10^3$ | 1187.79 | 0.00 | 0.00 | Segment Tree |
| 34 | $10^3$ | $10^4$ | 1187.97 | 0.00 | 15.63 | Segment Tree |
| 35 | $10^3$ | $10^5$ | 1203.22 | 0.00 | 0.00 | Segment Tree |
| 36 | $10^3$ | $10^6$ | 1188.35 | 453.22 | 281.24 | Sparse Table |
| 37 | $10^4$ | $10^3$ | 1188.31 | 0.00 | 0.00 | Segment Tree |
| 38 | $10^4$ | $10^4$ | 1203.63 | 15.64 | 15.63 | Sparse Table |
| 39 | $10^4$ | $10^5$ | 1203.54 | 0.00 | 0.00 | Segment Tree |
| 40 | $10^4$ | $10^6$ | 1187.84 | 515.62 | 312.86 | Sparse Table |
| 41 | $10^5$ | $10^3$ | 1187.84 | 15.62 | 0.00 | Sparse Table |
| 42 | $10^5$ | $10^4$ | 1188.02 | 0.00 | 15.63 | Segment Tree |
| 43 | $10^5$ | $10^5$ | 1187.65 | 0.00 | 0.00 | Segment Tree |
| 44 | $10^5$ | $10^6$ | 1203.31 | 531.25 | 312.48 | Sparse Table |
| 45 | $10^6$ | $10^3$ | 1187.83 | 109.37 | 703.11 | Segment Tree |
| 46 | $10^6$ | $10^4$ | 1187.83 | 125.23 | 703.18 | Segment Tree |
| 47 | $10^6$ | $10^5$ | 1203.20 | 124.99 | 703.12 | Segment Tree |
| 48 | $10^6$ | $10^6$ | 1187.82 | 718.85 | 1063.34 | Segment Tree |

**Table 3.** Algorithm's performance on tests with the length of each query being close 500.

| Test Index | N | M | SQR Decomposition | Segment Tree | Sparse Table | Best Solution |
|---|---|---|---|---|---|---|
| 49 | $10^3$ | $10^3$ | 1203.53 | 0.00 | 0.00 | Segment Tree |
| 50 | $10^3$ | $10^4$ | 1187.90 | 0.00 | 15.63 | Segment Tree |
| 51 | $10^3$ | $10^5$ | 1187.82 | 46.87 | 31.23 | Sparse Table |
| 52 | $10^3$ | $10^6$ | 1187.90 | 406.46 | 313.05 | Sparse Table |
| 53 | $10^4$ | $10^3$ | 1203.49 | 0.00 | 0.00 | Segment Tree |
| 54 | $10^4$ | $10^4$ | 1187.90 | 15.62 | 0.00 | Sparse Table |
| 55 | $10^4$ | $10^5$ | 1187.65 | 46.87 | 31.25 | Sparse Table |
| 56 | $10^4$ | $10^6$ | 1187.76 | 469.14 | 328.84 | Sparse Table |
| 57 | $10^5$ | $10^3$ | 1219.31 | 0.00 | 62.53 | Segment Tree |
| 58 | $10^5$ | $10^4$ | 1203.92 | 31.24 | 62.49 | Segment Tree |
| 59 | $10^5$ | $10^5$ | 1203.43 | 62.49 | 93.71 | Segment Tree |
| 60 | $10^5$ | $10^6$ | 1188.65 | 531.25 | 438.27 | Sparse Table |
| 61 | $10^6$ | $10^3$ | 1204.06 | 109.37 | 703.12 | Segment Tree |
| 62 | $10^6$ | $10^4$ | 1203.97 | 125.01 | 703.53 | Segment Tree |
| 63 | $10^6$ | $10^5$ | 1203.31 | 172.22 | 734.36 | Segment Tree |
| 64 | $10^6$ | $10^6$ | 1218.75 | 656.62 | 1078.96 | Segment Tree |

**Table 4.** Algorithm's performance on tests with the length of each query being close 5.

## 4 Conclusion

Conclusion: The Minimum Range Query (RMQ) problem has been well studied in the field of computer science and has many practical applications. Square Root Decomposition, Segment Trees, and Sparse Tables have been reviewed as solutions to the RMQ problem. The preprocessing and querying of each solution have different time complexity, and the performance of each solution differs based on the input data's specific characteristics.

A wide range of test inputs were used to implement and test the three solutions, and their performance was compared. Sparse Table was the fastest solution when the number of queries was small and the array was static, but it had a high preprocessing time compared to the other solutions, while Segment Tree performed better when the length of the queries was large. The square root decomposition method had an unexpectedly high processing time.

## References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd ed.). The MIT Press.
2. Author, Luis M. S. Russo. Range Minimum Queries in Minimal Space
3. Stanford CS116, `http://web.stanford.edu/class/cs166/`. Last accessed 19 December 2022
4. SQRT Decomposition, `https://cp-algorithms.com/data_structures/sqrt_decomposition.html`. Last accessed 3 January 2023
5. Segment Tree, `https://cp-algorithms.com/data_structures/segment_tree.html`. Last accessed 20 December 2022
6. Fischer J., Heun V. (2006) Theoretical and Practical Improvements on the RMQ Problem, with Applications to LCA and LCE. In: Lewenstein M., Valiente G. (eds) Combinatorial Pattern Matching. CPM 2006. Lecture Notes in Computer Science, vol 4009. Springer, Berlin, Heidelberg
7. Setiadi, Iskandar. (2012). Segment Tree for Solving Range Minimum Query Problems. 10.13140/2.1.4279.2643.