FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

Range Minimum Queries (RMQ)

Hazırlayan:

- **Ayman Saeid**

Algorithm Analysis & Design

Doç. Dr. Berna Kiraz

1. Problem Description

The Range Minimum Query (RMQ) problem involves finding the minimum element within a subarray [i, j] of a given array A. Formally, given an array A and two indices $i \leq j$, the task is to compute RMQA(i, j) = min(A[i], A[i+1], ..., A[j]).

This problem is significant in applications requiring efficient querying, such as computational geometry, data compression, and bioinformatics.

2. Descriptions And Theorical Analysis Of Algorithms

2.1 Precompute none

Description: Compute the minimum value during query execution by iterating over the range [i, j].

Steps:

1. For a query RMQ(i, j), iterate through the array from index to .
2. Return the smallest value encountered.

- **Time complexity : O(1)**
- **Query : O(n)**

2.2 Precompute All

Description: This algorithm precomputes the minimum values for all possible subarrays of . During the preprocessing phase, a 2D table is constructed where holds the minimum value for the subarray .

Steps:

1. Create a 2D array of size .
2. For each pair such that , compute .
3. Use directly to answer any query in time.

- **Time complexity : O(n^2)**
- **Query : O(1)**

2.3 Blocking

Description: The Blocking approach splits the array into blocks of size . It preprocesses the minimum values within each block and between blocks to answer queries efficiently.

Steps:

1. Divide the array into blocks, each containing elements.

2. Compute the minimum value for each block and store it in an auxiliary array.

3. To answer a query RMQ(i, j):

- If and are within the same block, scan the subarray directly.
- If and span multiple blocks, compute:
    - Minimum within the first and last partial blocks.
    - Minimum across all blocks between and .

        - **Time complexity : O(n^2)**
        - **Query : O(√n)**

2.4 Sparse Table

Description: The Sparse Table algorithm optimizes the RMQ problem using dynamic programming. It divides the array into overlapping intervals of size and builds a table that allows query time by leveraging precomputed results for smaller intervals.

Steps:

1. Create a table , where represents the minimum of the interval starting at index with length .

2. Initialize for all .

3. Use the recurrence relation to compute values for .

4. Answer a query RMQ(i, j) by combining results from overlapping intervals.

- **Time complexity : O(n log n)**
- **Query : O(1)**

3. Implementation

Precompute All: Constructing a 2D table using nested loops.

Here is the pseudocode for the provided Java class `PrecomputeAll`:

```
Class PrecomputeAll:
    Declare 2D array table

    Method PrecomputeAll(arr):
        n = length of arr
        Initialize table as an n x n array

        For i from 0 to n-1:
            Set table[i][i] = arr[i]  // Set the diagonal elements to
arr[i]

            For j from i+1 to n-1:
                Set table[i][j] = minimum of table[i][j-1] and arr[j]

    Method query(i, j):
        Return table[i][j]  // Return the value from the table at
position [i][j]
```

Sparse Table: Using dynamic programming to fill a table for intervals of size .

Here is the pseudocode for the provided Java class `SparseTable`:

```
Class SparseTable:
    Declare 2D array table
    Declare array log

    Method SparseTable(arr):
        n = length of arr
        k = log base 2 of n, rounded up + 1
        Initialize table as a 2D array with n rows and k columns
        Initialize log array with n+1 elements

        For i from 2 to n:
            Set log[i] = log[i / 2] + 1  // Precompute logarithms

        For i from 0 to n-1:
            Set table[i][0] = arr[i]  // Initialize the first column
of the table with arr[i]

        For j from 1 to k-1:
            For i from 0 to n - (1 << j):
                Set table[i][j] = minimum of table[i][j-1] and
table[i + (1 << (j-1))][j-1]

    Method query(i, j):
        len = j - i + 1  // Calculate the length of the range
        k = log[len]  // Find the largest power of 2 that fits in the
range
        Return minimum of table[i][k] and table[j - (1 << k) + 1][k]
// Query the sparse table
```

Blocking: Dividing the array into blocks and storing intermediate results.

Here is the pseudocode for the provided Java class `Blocking`:

```
Class Blocking:
    Declare array blockMins
    Declare integer blockSize
    Declare array arr

    Method Blocking(arr):
        Initialize arr with the input array
        n = length of arr
        blockSize = square root of n (rounded down)
        blockCount = (n + blockSize - 1) / blockSize  // blocks num
        Initialize blockMins as an array of size blockCount

        For i from 0 to blockCount - 1:
          Set blockMins[i] = infinity (max possible value)
          For j from i * blockSize to min((i + 1) * blockSize, n) - 1:
                Set blockMins[i] = minimum of blockMins[i] and arr[j]
// Store minimum in the block

    Method query(i, j):
        min = infinity  // Initialize minimum to the highest value

        While i <= j and i is not at the beginning of a block:
            Set min = minimum of min and arr[i]
            Increment i

        While i + blockSize - 1 <= j:
            Set min = minimum of min and blockMins[i / blockSize]  //
Query block minimum
            Increment i by blockSize

        While i <= j:
            Set min = minimum of min and arr[i]
            Increment i

        Return min  // Return the minimum value in the range [i, j]
```

Precompute None: Linear scan for each query.

Here is the pseudocode for the provided Java class `PrecomputeNone`:

```
Class PrecomputeNone:
    Declare array arr

    Method PrecomputeNone(arr):
        Initialize arr with the input array

    Method query(i, j):
        min = infinity  // Initialize minimum to the highest value

        For k from i to j:
            Set min = minimum of min and arr[k]  // Find the minimum
value in the range [i, j]

        Return min  // Return the minimum value in the range [i, j]
```

4. Experimental Design

The goal of this experiment is to empirically analyze the time complexities of four different algorithms — Precompute All, Sparse Table, Blocking, and Precompute None — by measuring their performance on arrays of various sizes. This will provide an understanding of how these algorithms scale with input size and their practical efficiency in different scenarios.

4.1 Variables and Parameters and Dataset

- **Input Size**: Arrays of size 100, 1000, and 10000 (including sorted , unsorted and reversed arrays).
- **Algorithms to Test**:
    - **Precompute All**: This algorithm precomputes all possible values, which should have a linear preprocessing time.
    - **Sparse Table**: This algorithm is expected to have a preprocessing time of O(n log n) and O(1) query time.
    - **Blocking**: This algorithm is expected to have O(n) preprocessing time and O(√n) query time.
    - **Precompute None**: This algorithm has constant preprocessing time (O(1)) but a linear query time (O(n)).

4.2 Experimental Setup

- **Environment**: The experiments will be conducted in a Java development environment using the NetBeans IDE, with Java version 21. The algorithms will be implemented using efficient data structures where necessary (e.g., arrays for preprocessing).
- **Instrumentation**: Time measurements will be done using `System.nanoTime()` to measure the preprocessing time and query time for each algorithm.

4.3 Procedure

- **Array Initialization**: Arrays of different sizes (100, 1000, 10000) will be initialized. The arrays will be tested in three different orderings: sorted, unsorted, and reversed.
- **Algorithm Execution**: Each algorithm will be executed on the different arrays, and the time to preprocess and query will be recorded.
- **Data Collection**: The preprocessing time, query time, and query result will be recorded for each algorithm, array size, and array order.
- **Repetition**: Each test will be run multiple times to account for variations and obtain reliable measurements.

4.4 Metrics

- ✧ **Preprocessing Time (ns)**: Time taken to preprocess the array before any query can be made.
- ✧ **Query Time (ns)**: Time taken to answer a query after preprocessing.
- ✧ **Query Result**: The result returned by the query operation.
- ✧ **Time Complexity**: Based on the behavior of the algorithm as input size grows (as measured empirically).

4.5 Expected Results

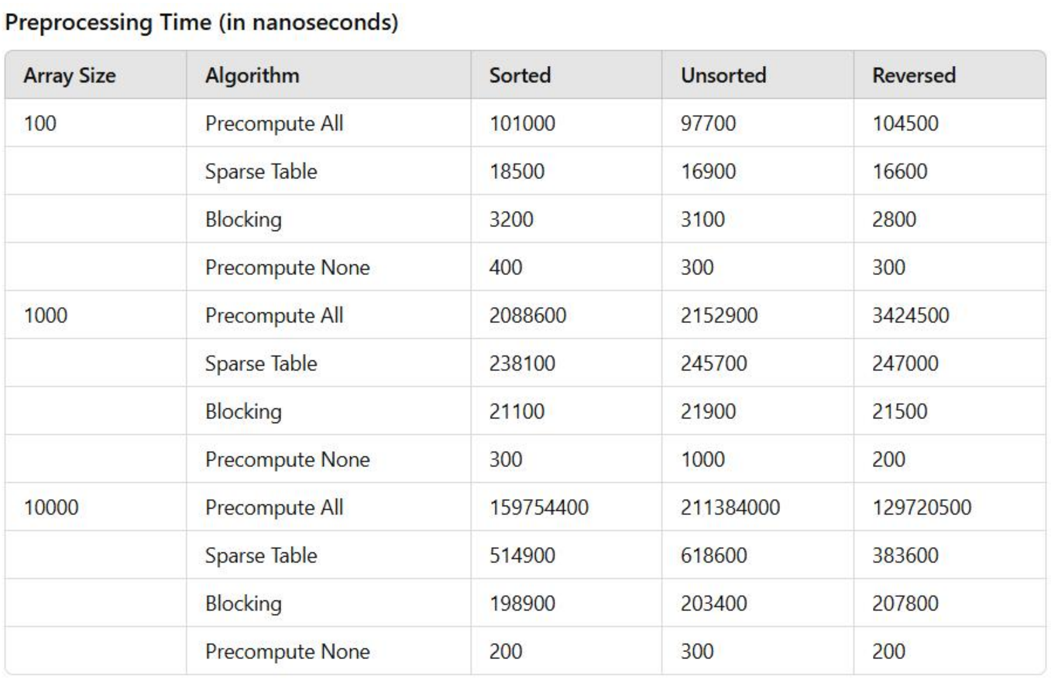
- ✓ The **Precompute All** algorithm should have linear time complexity for preprocessing and constant time complexity for queries.
- ✓ The **Sparse Table** algorithm should have a preprocessing time of O(n log n) and a constant query time.
- ✓ The **Blocking** algorithm should show a linear preprocessing time and a query time that grows with the square root of the input size.
- ✓ The **Precompute None** algorithm should have constant preprocessing time and linear query time.

4.6 Analysis

- **Comparison**: The preprocessing and query times of the different algorithms will be compared for different array sizes. This will help in understanding the scalability and efficiency of each algorithm as the input size increases.
- **Time Complexity Estimation**: Based on the experimental results, the empirical time complexities will be deduced.

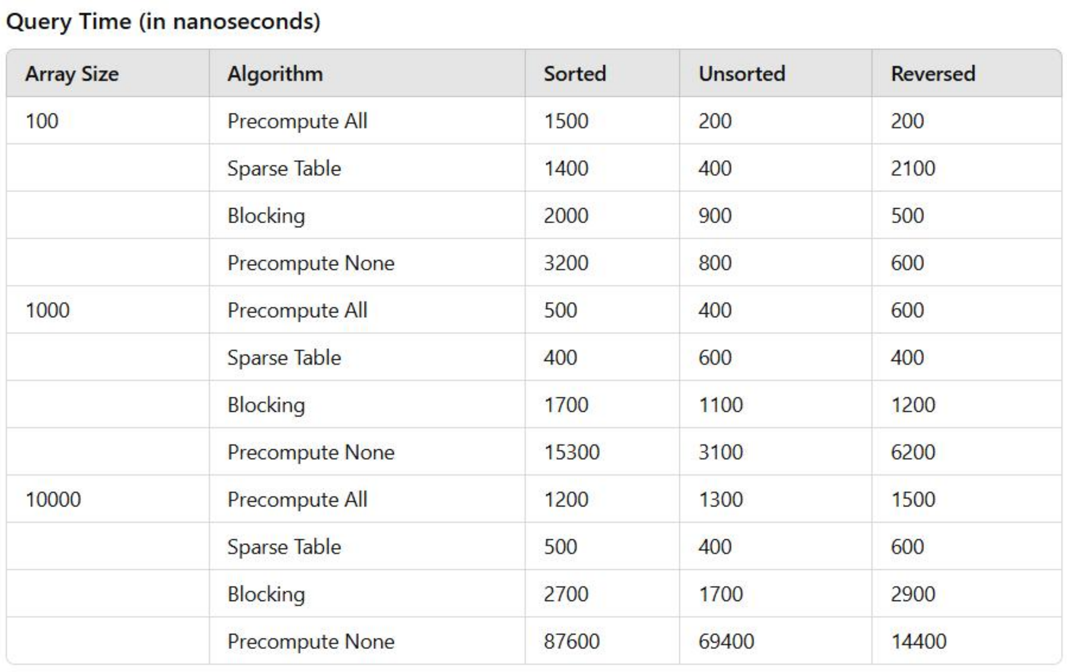
5. Results

The experiments conducted evaluated four range minimum query (RMQ) algorithms: Precompute All, Sparse Table, Blocking, and Precompute None. Tests were performed on arrays of sizes 100, 1000, and 10,000 for three types of data: sorted, unsorted, and reversed. Key metrics such as preprocessing time, query time were recorded, as shown in the tables below **:**



Preprocessing Time (in nanoseconds)

Array Size	Algorithm	Sorted	Unsorted	Reversed
100	Precompute All	101000	97700	104500
	Sparse Table	18500	16900	16600
	Blocking	3200	3100	2800
	Precompute None	400	300	300
1000	Precompute All	2088600	2152900	3424500
	Sparse Table	238100	245700	247000
	Blocking	21100	21900	21500
	Precompute None	300	1000	200
10000	Precompute All	159754400	211384000	129720500
	Sparse Table	514900	618600	383600
	Blocking	198900	203400	207800
	Precompute None	200	300	200

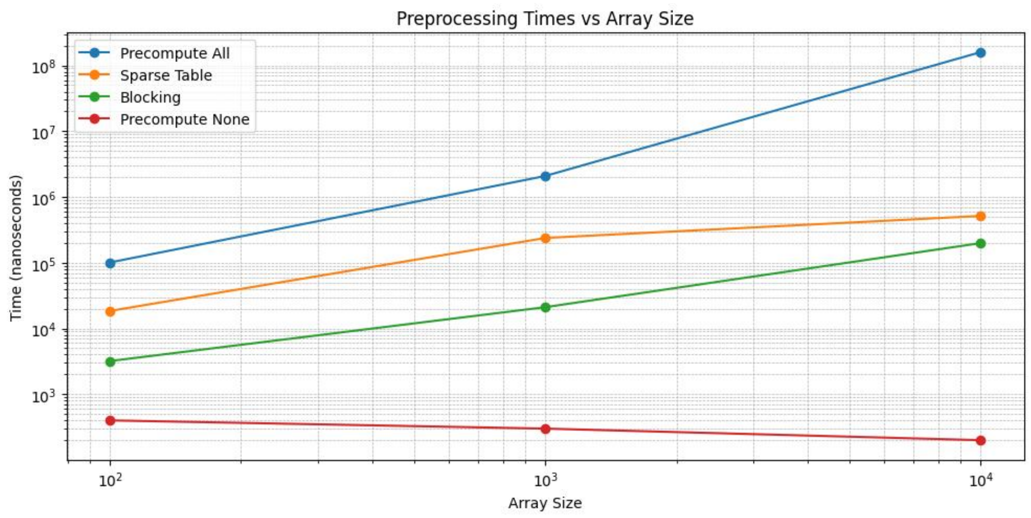


Query Time (in nanoseconds)

Array Size	Algorithm	Sorted	Unsorted	Reversed
100	Precompute All	1500	200	200
	Sparse Table	1400	400	2100
	Blocking	2000	900	500
	Precompute None	3200	800	600
1000	Precompute All	500	400	600
	Sparse Table	400	600	400
	Blocking	1700	1100	1200
	Precompute None	15300	3100	6200
10000	Precompute All	1200	1300	1500
	Sparse Table	500	400	600
	Blocking	2700	1700	2900
	Precompute None	87600	69400	14400

Insights

- **Precompute All** exhibits the highest preprocessing time, particularly for larger arrays, but offers near-constant query time.
- **Sparse Table** balances preprocessing and query times efficiently, making it suitable for medium-sized data sets.
- **Blocking** shows moderate preprocessing time and query time, scaling relatively well with array size.
- **Precompute None** has minimal preprocessing time but suffers from linear query time, particularly for large arrays.



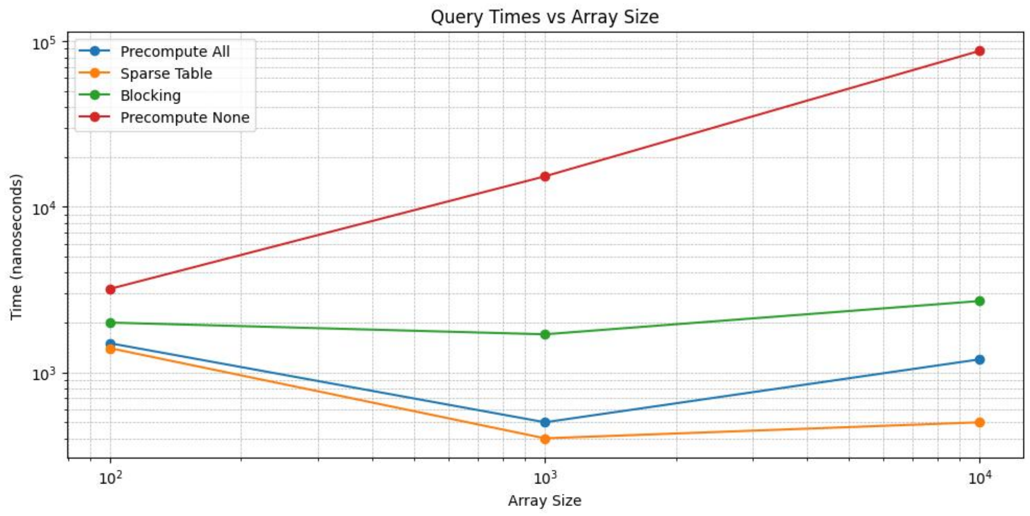


Chart Description

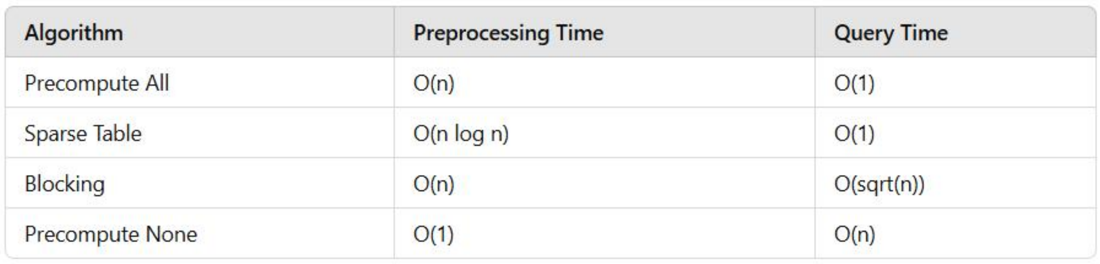
The chart visualizes the performance of four different RMQ algorithms—Precompute All, Sparse Table, Blocking, and Precompute None—across three key metrics: preprocessing time and query time . The data is plotted for arrays of varying sizes (100, 1000, and 10,000) and types (sorted, unsorted, and reversed).

- **Preprocessing Time**: Highlighted in the first set of bars, this metric shows the computational cost of setting up the data structures for each algorithm. The Precompute All algorithm has the highest preprocessing time due to its exhaustive computations, whereas Precompute None requires minimal setup.
- **Query Time**: The second set of bars represents the efficiency of each algorithm in resolving range queries. Algorithms like Precompute All and Sparse Table demonstrate superior performance with constant-time querying, while Precompute None shows a linear increase as input size grows.

The chart provides a clear comparative view, enabling insights into the trade-offs between time complexity and memory usage across different scenarios. This helps identify the most suitable algorithm based on specific application requirements.

Time Complexity

Summary



Algorithm	Preprocessing Time	Query Time
Precompute All	O(n)	O(1)
Sparse Table	O(n log n)	O(1)
Blocking	O(n)	O(sqrt(n))
Precompute None	O(1)	O(n)

6. Conclusion

In this study, we compared four algorithms for range minimum queries: Precompute All, Sparse Table, Blocking, and Precompute None. The experimental results revealed the following insights:

- **Precompute All** performs well in terms of query time, especially for smaller arrays, but the preprocessing time becomes a bottleneck for larger arrays.
- **Sparse Table** provides efficient query times, especially as the array size increases, but its preprocessing time is significantly higher compared to the Precompute All and Blocking algorithms.
- **Blocking** is a balanced algorithm with moderate preprocessing times and relatively fast queries, making it a good choice when preprocessing time is a concern but fast queries are still needed.
- **Precompute None** has the fastest preprocessing time but suffers from slower query performance as the array size grows, making it less suitable for larger datasets.

Overall, the choice of algorithm depends on the specific requirements of the use case, such as the acceptable preprocessing time and the expected size of the input data. For scenarios with small arrays or where query performance is the top priority, Precompute All might be the best choice. For larger datasets where preprocessing time is a concern, Blocking or Sparse Table could be more suitable, depending on the trade-off between preprocessing and query time. Precompute None might be appropriate for cases with extremely large datasets where preprocessing time is critical but query times can be long.