FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

Student:

Name: Enes

Surname: Karaçay

ID Number: 2121221047

Department: Computer Enginnering

Project:

Topic: Range Minimum Query (RMQ)

Course:

Name: Algorithm Analysis and Design

Instructor: Berna Kiraz

Content

1. PROBLEM DESCRIPTION

1.1 Introduction to Range Minimum Query (RMQ)

The Range Minimum Query (RMQ) is a fundamental problem in computer science that involves finding the minimum value within a specified range of an array. It is widely used in applications such as sorting, data analysis, and algorithm design. This section provides a formal definition of RMQ, explains how it works, discusses its applications, and presents an example to illustrate its operation.

1.1.1 Formal Definition of RMQ

The RMQ problem is defined as follows:

- Given an array **A[1...n]**, the operation **RMQ(i, j)** finds the minimum value in the subarray **A[i...j]** ($1 \leq i \leq j \leq n$).

The goal is to process queries efficiently, especially when the number of queries is large or the array is extensive. To achieve this, RMQ solutions often involve preprocessing the array to build auxiliary data structures.

1.1.2 How RMQ Works

The RMQ problem is typically addressed in two stages:

1.1.2.1 Preprocessing

In this stage, the array is analyzed to prepare auxiliary data structures that allow for efficient query resolution. The preprocessing time and memory usage depend on the chosen approach. The trade-off between preprocessing complexity and query efficiency is a critical factor in RMQ algorithms.

1.2.2.2 Query Processing

Once the preprocessing is complete, the prepared data structures are used to answer queries such as **RMQ(i, j)** quickly. Efficient query resolution ensures that results are obtained with minimal computational overhead, even for large arrays.

1.1.3 Applications of RMQ

RMQ has numerous applications in computer science and related fields. Some notable examples include:

1.1.3.1 Array Processing and Segment Analysis

RMQ is used to analyze properties of array segments, such as finding the minimum value within a given range. For example, in stock market analysis, RMQ can identify the lowest price within a specific time interval.

1.1.3.2 Lowest Common Ancestor (LCA)

In tree structures, RMQ can determine the lowest common ancestor of two nodes by mapping the tree to an array and applying RMQ techniques.

1.1.3.3 Text Processing

RMQ-based algorithms accelerate substring search and comparison operations in text data, such as finding the smallest lexicographical substring.

1.1.3.4 Dynamic Programming

RMQ simplifies and speeds up dynamic programming problems that involve interval-based calculations.

1.1.4 Example of RMQ

Example Array: A = [3, 5, 2, 7, 9, 1, 6]

- **Query 1: RMQ(2, 5):** Find the minimum value between indices 2 and 5. Subarray A[2...5] = [5, 2, 7, 9]. The minimum value is **2**.
- **Query 2: RMQ(4, 7):** Find the minimum value between indices 4 and 7. Subarray A[4...7] = [7, 9, 1, 6]. The minimum value is **1**.

1.1.5 Challenges and Solutions in RMQ

The primary challenge in RMQ lies in balancing preprocessing time, memory usage, and query speed. Key considerations include:

- **Naive Approach:** Directly examine all elements in the queried range to find the minimum. This approach has a query time complexity of O(n).
- **Optimized Approaches:** Advanced algorithms, such as the Sparse Table method, reduce query time to O(1) by using preprocessing. These methods trade higher preprocessing time and memory for faster query resolution.

Efficient RMQ solutions require selecting an approach that suits the specific problem constraints and requirements.

2. ALGORITHMS:

2.1 Precompute All: A Detailed Explanation

The Precompute All approach is a brute-force method for solving the Range Minimum Query (RMQ) problem. It involves computing and storing the minimum values for all possible ranges in the array during a preprocessing phase. This ensures that any query can be answered in constant time (O(1)) by simply looking up a precomputed table. Below is a detailed explanation of this approach, including steps, examples, and theoretical performance analysis.

2.1.1 Steps to Implement Precompute All

1. Input Array Start with an input array A[0…n−1], where n is the size of the array.
2. Preprocessing Create a 2D table M[i][j], where M[i][j] represents the minimum value in the subarray A[i…j]. Here, i and j are indices such that $0 \leq i \leq j < n$. Fill the table by iterating over all possible pairs of indices (i,j): If i=j: M[i][j]=A[i], since the range contains a single element. If i<j: M[i][j]=min(A[i],A[i+1],…,A[j]).
3. Query Phase For a query (i,j), directly return M[i][j] as the result.

2.1.2 Example: Precomputing the Table

Given an array A=[4,2,6,1], the goal is to preprocess all possible ranges.

Step 1: Initialize the Table

- For i=j:

M[0][0]=4, M[1][1]=2, M[2][2]=6, M[3][3]=1.

Step 2: Compute for Ranges

- M[0][1]=min(A[0],A[1])=min(4,2)=2.
- M[0][2]=min(A[0],A[1],A[2])=min(4,2,6)=2.
- M[0][3]=min(A[0],A[1],A[2],A[3])=min(4,2,6,1)=1.
- M[1][2]=min(A[1],A[2])=min(2,6)=2.
- M[1][3]=min(A[1],A[2],A[3])=min(2,6,1)=1.
- M[2][3]=min(A[2],A[3])=min(6,1)=1.

Step 3: Final Table

The resulting 2D table M[i][j] is as follows

$$M = \begin{bmatrix} 4 & 2 & 2 & 1 \\ - & 2 & 2 & 1 \\ - & - & 6 & 1 \\ - & - & - & 1 \end{bmatrix}$$

Table 2.1(PrecomputeAll Matrix)

Here, M[i][j] represents the minimum value in the range A[i…j].

Query Example

- For RMQ(1,3): Look at M[1][3]=1.
- For RMQ(0,2): Look at M[0][2]=2.

2.1.3 Theoretical Performance Analysis

1. **Preprocessing Time Complexity**

- Computing M[i][j] for all $i \leq j$ involves $n(n+1)/2$ ranges.
- For each range, finding the minimum value takes $O(j-i+1)$.
- The total preprocessing time is $O(n^2)$.

2. **Query Time Complexity**

- Each query is resolved in constant time by directly accessing M[i][j]. The query time complexity is O(1).

3. **Space Complexity**

- The table M requires $O(n^2)$ space to store all ranges.

Advantages

- **Fast Queries:** Any query can be answered in O(1), making it ideal for applications with frequent queries.
- **Simple Implementation:** The logic is straightforward and easy to code.

Disadvantages

- **High Space Requirement:** For large arrays, the $O(n^2)$ space requirement becomes impractical.
- **High Preprocessing Time:** The preprocessing step takes $O(n^2)$, which may be infeasible for large datasets.

When to Use

The Precompute All approach is suitable for scenarios where:

- The input array is small (n is small enough to handle $O(n^2)$ space).
- Query speed is the top priority, and preprocessing time is not a concern.

2.2 Sparse Table: A Detailed Explanation

The **Sparse Table** approach is an efficient method for solving the **Range Minimum Query (RMQ)** problem. This method preprocesses the data in O(nlogn) time and answers queries in O(1) time. However, it's only suitable for static arrays, meaning the input array cannot be updated after preprocessing. Below is a detailed explanation of this approach, including steps, examples, theoretical performance analysis, and usage scenarios.

2.2.1 Steps to Implement Sparse Table

- **Input Array**: Start with an input array A[0…n−1], where nnn is the size of the array.
- **Preprocessing**:
  - Create a 2D table ST[i][j], where ST[i][j] represents the minimum value in the subarray A[i…$i+2^j-1$].
  - Here, i is the starting index, and j is the power of 2 (logarithmic scale).
  - Fill the table as follows:
    - **Base Case** (j=0): ST[i][0]=A[i], since a single-element range is the element itself.
    - **Recursive Case** (j>0): ST[i][j]=min(ST[i][j−1],ST[i+2j−1][j−1]).
- **Query Phase**:
  - For a query (L,R) calculate the length of the range: length=length=R−L+1.
  - Compute the largest power of 2 that fits into the range: $k=[\log_2(length)]$.
  - Find the minimum value: result=min(ST[L][k], ST[R -2^k + 1][k]).

2.2.2 Example: Precomputing the Sparse Table

Input Array: A=[4,2,6,1,5,7].

Step 1: Base Case (j=0)

ST[i][0]=A[i].

Table:

$$ST = \begin{bmatrix} 4 & - & - & - \\ 2 & - & - & - \\ 6 & - & - & - \\ 1 & - & - & - \\ 5 & - & - & - \\ 7 & - & - & - \end{bmatrix}$$

:

Table 2.2 (Sparse Table Matrix)

Step 2: Compute for j=1

ST[i][1] = min(ST[i][0],ST[i+1][0])

- ST[0][1]=min(4,2)=2
- ST[1][1]=min(2,6)=2.
- ST[2][1]=min(6,1)=1.
- ST[3][1]=min(1,5)=1.
- ST[4][1]=min(5,7)=5.

Table 2.3 (Sparse Table Matrix)

$$ST = \begin{bmatrix} 4 & 2 & - & - \\ 2 & 2 & - & - \\ 6 & 1 & - & - \\ 1 & 1 & - & - \\ 5 & 5 & - & - \\ 7 & - & - & - \end{bmatrix}$$

Step 3: Compute for j=2
ST[i][2]=min(ST[i][1],ST[i+2^1][1])

- ST[0][2]=min(2,1)=1.
- ST[1][2]=min(2,1)=1.
- ST[2][2]=min(1,5)=1.

Table:

$$ST = \begin{bmatrix} 4 & 2 & 1 & - \\ 2 & 2 & 1 & - \\ 6 & 1 & 1 & - \\ 1 & 1 & - & - \\ 5 & 5 & - & - \\ 7 & - & - & - \end{bmatrix}$$

Table 2.4 (Sparse Table Matrix)

2.2.3 Query Example

Query 1: RMQ(1,4)

- length=4−1+1=4, k=⌊log2(4)⌋=2.
- result=min(ST[1][2])=1.

Query 2: RMQ(2,5)

- length=5−2+1=4, k=⌊log2(4)⌋=2.
- result=min(ST[2][2])=1.

2.2.4 Theoretical Performance Analysis

1. **Preprocessing Time Complexity**

- O(n logn), as each j-level involves O(n) work, and there are logn levels.

2. **Query Time Complexity**

- O(1), since it involves only two lookups and one comparison.

3. **Space Complexity**

- O(n logn), as the table has nnn rows and logn columns.

2.2.5 Advantages and Disadvantages

Advantages:

- Extremely fast queries (O(1)).
- Lower memory usage compared to Precompute All (O(nlogn) vs $O(n^2)$).

Disadvantages:

- Only works for static arrays; updates are not supported.

2.2.6 When to Use

- **Static Arrays:** The input array remains unchanged after preprocessing.

- **Frequent Queries:** Applications where the number of queries is significantly higher than the preprocessing overhead.

- **Moderately Large Arrays:** Arrays that are too large for $O(n^2)$ space but manageable for O(nlogn). Sparse Table is less suitable for dynamic scenarios where the input array is frequently updated.

2.3 Blocking: A Detailed Explanation

The Blocking approach for solving the Range Minimum Query (RMQ) problem is an optimization technique that divides the input array into blocks of fixed size. This technique aims to balance preprocessing time, query time, and space complexity, achieving a practical solution for handling larger datasets. The Blocking approach is particularly useful when dealing with very large static arrays, where the preprocessing time and space complexity need to be carefully optimized.

2.3.1 Steps to Implement Blocking

1. **Input Array**: Start with an input array A[0…n−1] where n is the size of the array.
2. **Preprocessing**:

- Divide the array into blocks of size B=⌈√n⌉. The block size is chosen as the square root of the array size to balance the cost of preprocessing and query time.
- For each block Bi, compute and store the minimum value for all elements in that block. This can be done in O(B) time for each block.
- Create a secondary array M[], where each entry M[i] holds the minimum value for the block Bi. The array M[] has O(√n) entries.
- The remaining structure is the preprocessing phase for queries that span across blocks. We need to calculate the minimum value for the remaining array elements that do not fall within complete blocks.

3. **Query Phase**:

- For a query (L,R), determine which blocks the indices L and R fall into.
- If L and R fall within the same block, simply compute the minimum value in that range using a linear scan (which takes O(B) time).
- If L and R fall in different blocks, do the following:
- Compute the minimum value from the elements between L and the end of its block.
- Compute the minimum value from the elements between the start of the block containing R and R.
- For all blocks completely between L and R, the minimum value can be obtained in O(1) time from the precomputed values in M[].

2.3.2 Example: Precomputing for Blocking

Let’s consider the array A=[4,2,6,1,5,7,8,9] with n=8.

Step 1: Divide into blocks

- The block size is B=⌈√8⌉=3, so we divide the array into blocks of size 3:
  - Block 1: [4,2,6]
  - Block 2: [1,5,7]
  - Block 3: [8,9]

Step 2: Precompute minimums for each block

- Block 1: Minimum = 2
- Block 2: Minimum = 1

- Block 3: Minimum = 8

Now, construct the array M[]=[2,1,8] where each entry represents the minimum of the corresponding block.

Step 3: Query Example

For a query RMQ(1,7) we proceed as follows:

- Block containing index 1: [2,6] (query starts at index 1, so minimum value from A[1] to A[2] is 2).
- Block containing index 7: [8,9] (query ends at index 7, so minimum value from A[6] to A[7] is 8).
- Blocks in between: [1,5,7], where the precomputed minimum value is 1 from array M[].

Thus, the result is min(2,1,8)=1.

2.3.3 Theoretical Performance Analysis

1. **Preprocessing Time Complexity**:
   - Computing the minimum for each block takes O(B) time, and there are n/B blocks. Therefore, the time complexity for preprocessing is O(n).
   - The space complexity is O(n/B)+O(√n)=O(√n) because the space for the blocks and the secondary array M[] are both O(√n).
2. **Query Time Complexity**:
   - For a query, the number of blocks spanned is O(√n), and we perform constant-time lookups for fully spanned blocks and O(B) time for partial blocks. Therefore, the query time complexity is O(√n).
3. **Space Complexity**:
   - The space complexity is O(n), as we store the array A[] and the auxiliary array M[].

2.3.4 Advantages and Disadvantages

Advantages:

- **Balanced Time Complexity**: The preprocessing time is O(n), and the query time is O(√n), making it a good compromise between speed and memory usage for large static arrays.
- **Memory Efficiency**: The space complexity is lower than $O(n^2)$ methods like Precompute All while still providing efficient queries.

Disadvantages:

- **Suboptimal for Dynamic Arrays**: The Blocking approach does not handle updates efficiently. If the input array changes, the entire preprocessing step needs to be redone, making it unsuitable for dynamic arrays.

2.3.5 When to Use

The Blocking approach is most effective in the following scenarios:

- **Large Static Arrays**: When dealing with large arrays where both preprocessing time and space complexity are a concern, but the array will not change frequently.
- **Moderate Query Speed Requirements**: When the need for fast queries is moderate and the primary goal is efficient space usage.
- **Large, Sparse Data**: In cases where the input array is large and sparse, the blocking technique ensures that memory is utilized effectively while queries remain efficient.

2.4 Precompute None: A Detailed Explanation

The **Precompute None** approach for solving the Range Minimum Query (RMQ) problem is a strategy where no preprocessing is done before answering the queries. Instead, each query is handled by directly scanning the array within the given range to find the minimum value. This approach is the simplest but also the least efficient in terms of query time, as it requires examining the array elements for every query. Below is a detailed explanation of the approach, including steps, examples, and theoretical performance analysis.

2.4.1 Steps to Implement Precompute None

1. **Input Array**: Start with an input array A[0…n−1], where n is the size of the array.
2. **Query Phase**:

- For each query RMQ(L,R), directly scan the array from index L to index R to find the minimum value in that range.
- This can be done by iterating over all elements in the range and keeping track of the smallest element encountered.

2.4.2 Example: Querying Without Precomputation

Let's consider the array A=[4,2,6,1,5,7] with n=6. We will demonstrate how to answer a range minimum query using the **Precompute None** approach.

Query 1: RMQ(1, 4)

- For this query, we need to find the minimum value in the subarray A[1]…A[4], which corresponds to the array slice [2,6,1,5].
- We directly scan through the array slice:
- Start with the first element, 2. Keep track of the minimum value.
- Compare with 6, the new minimum is 2.
- Compare with 1, the new minimum is 1.
- Compare with 5, the minimum stays 1.
- The result of the query is 1.

Query 2: RMQ(2, 5)

- For this query, we need to find the minimum value in the subarray A[2]…A[5], which corresponds to the array slice [6,1,5,7].
- We directly scan through the array slice:
- Start with the first element, 6. Keep track of the minimum value.
- Compare with 1, the new minimum is 1.
- Compare with 5, the minimum stays 1.
- Compare with 7, the minimum stays 1.
- The result of the query is 1.

2.4.3 Theoretical Performance Analysis

1. **Preprocessing Time Complexity**:

- **Precompute None** involves no preprocessing. Therefore, the preprocessing time complexity is O(1), meaning no time is spent before answering queries.

2. **Query Time Complexity**:

- For each query, the time complexity is O(R−L+1), as we need to scan through the entire range [L,R] to find the minimum value. In the worst case, this is O(n) if the query range spans the entire array.

  Thus, the query time complexity is **linear in the size of the range**. For a query that covers the entire array, the time complexity becomes O(n).

3. **Space Complexity**:

- Since no additional space is required for preprocessing or storing auxiliary data structures, the space complexity is O(1). Only the input array is used.

2.4.4 Advantages and Disadvantages

Advantages:

- **Simple Implementation**: The Precompute None approach is extremely easy to implement. There are no complex data structures or preprocessing steps involved.
- **No Space Requirement**: Since no preprocessing is required, the space complexity is minimal, requiring only O(1) additional space (aside from the input array).

Disadvantages:

- **Slow Query Time**: For each query, we must scan the entire range from L to R to find the minimum, leading to a query time complexity of O(n) in the worst case. This is highly inefficient for large arrays with many queries.
- **Not Suitable for Frequent Queries**: If the number of queries is large, this approach becomes impractical due to the high time complexity for each query.

2.4.5 When to Use

The Precompute None approach is best suited for scenarios where:

- **Sparse Queries**: If there are only a few queries and the size of the array is not too large, this approach might be feasible. It avoids the overhead of preprocessing and space usage.
- **Small Arrays**: For small datasets, where the cost of scanning each range is minimal, this approach can be acceptable, but only when queries are infrequent.
- **Minimal Memory Use**: When memory constraints are extremely tight, the Precompute None approach is a good option, as it requires no additional storage for preprocessing.

Summary of Theoretical Performance

- This table, **Table 1**, presents the theoretical performance analysis of four different algorithms used to solve the Range Minimum Query (RMQ) problem. The algorithms are compared based on query time complexity, preprocessing time complexity, and memory complexity.
- **Table 1** provides a systematic overview of each algorithm's strengths and weaknesses, making it easier to understand their advantages and disadvantages. It serves as a general reference for evaluating theoretical performance.

Approach	Query Time Complexity	Preprocessing Time Complexity	Memory Complexity
Precompute None	$O(n)$ per query	$O(1)$	$O(n)$
Blocking	$O(n)$	$O(n)$	$O(n)$
Sparse Table	$O(1)$	$O(n \log n)$	$O(n \log n)$
Precompute All	$O(1)$	$O(n^2)$	$O(n^2)$

Table 1. (Theoretical Performance Comparison of Four Algorithms)

3. Implementation

- This section provides the pseudocode for each of the four algorithms used to solve the Range Minimum Query (RMQ) problem. Additionally, the step-by-step functionality of each algorithm is explained. The pseudocode simplifies the understanding of the algorithms' logic and their implementation.

3.1 Precompute All Algorithm Pseudocode

Explanation:

- The Precompute All algorithm calculates the minimum values for all possible subarrays in advance and stores them in a table. This allows any query to be answered in constant time, O(1). However, its preprocessing time complexity of $O(n^2)$ makes it computationally expensive for large datasets.

Pseudocode:

```
def PrecomputeAll(A, n):
    M = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(i, n):
            if i == j:
                M[i][j] = A[i]
            else:
                M[i][j] = min(M[i][j - 1], A[j])
    return M
def Query(M, i, j):
    return M[i][j]
```

Step-by-Step Explanation:

1. **Table Creation:**
   The algorithm creates a n×n table (M). This table stores the minimum values for all possible subarrays in the given array.
2. **Single-Element Subarrays:**
   If the subarray contains only one element (i==j), the minimum value is simply the element itself (A[i]).
3. **Minimum of Subarrays:**
   For larger subarrays, the minimum value is calculated by comparing the minimum of the previous subarray (M[i][j−1]) with the current element (A[j]).
4. **Query Answering:**
   Using the precomputed table, any query ([i,j]) can be answered in O(1) time by directly accessing the stored result.

Comments:

- Preprocessing time complexity: $O(n^2)$
- Query time complexity: O(1)
- Memory complexity: $O(n^2)$
- This algorithm is ideal for scenarios with a very high number of queries on a relatively small dataset.

3.2 Sparse Table Algorithm Pseudocode

Explanation:
The Sparse Table algorithm is particularly effective for static arrays. It preprocesses the array in O(nlogn)time and answers queries in O(1) time. It is well-suited for scenarios where updates to the array are not required.

Pseudocode:

```
def PrecomputeSparseTable(A, n):
    logn = int(math.log2(n)) + 1
    ST = [[0 for _ in range(logn)] for _ in range(n)]
        for i in range(n):
        ST[i][0] = A[i]
    for j in range(1, logn):
        for i in range(n - (1 << j) + 1):
            ST[i][j] = min(ST[i][j - 1], ST[i + (1 << (j - 1))][j - 1])
    return ST
def Query(ST, L, R):
    k = int(math.log2(R - L + 1))
    return min(ST[L][k], ST[R - (1 << k) + 1][k])
```

Step-by-Step Explanation:

1. **Table Preparation:**
   A n×logn table (ST) is created. This table stores the minimum values for subarrays of different lengths.
2. **Single-Element Intervals:**
   The first column (j=0) represents subarrays of size 1. For these intervals, the minimum value is the element itself (A[i]).

3. **Growing Subarrays:**
   For larger subarrays, the minimum is calculated by comparing two overlapping intervals from the previous column: ST[i][j−1] and ST[$i+2^{j-1}$][j−1].
4. **Query Answering:**
   For a query over the range [L,R], the range is divided into two intervals of size 2^k, where $k=\lfloor \log_2(R-L+1) \rfloor$. The result is the minimum of these two intervals.

Comments:

- Preprocessing time complexity: O(nlogn)
- Query time complexity: O(1)
- Memory complexity: O(nlogn)
- This algorithm is highly efficient for static datasets with frequent queries.

3.3 Blocking Algorithm Pseudocode

Explanation:
The Blocking algorithm divides the array into smaller blocks of approximately equal size. Minimum values for each block are precomputed. Queries that overlap blocks are computed by combining block-level and intra-block computations.

Pseudocode:

```
def PrecomputeBlocking(A, n):
    blockSize = math.ceil(math.sqrt(n))
    numBlocks = math.ceil(n / blockSize)
    B = [float('inf')] * numBlocks
    for i in range(n):
        blockIndex = i
        B[blockIndex] = min(B[blockIndex], A[i])
    return B, blockSize
def QueryBlocking(A, B, blockSize, L, R):
    minVal = float('inf')
    while L <= R:
        if L % blockSize == 0 and L + blockSize - 1 <= R:
            blockIndex = L
            minVal = min(minVal, B[blockIndex])
```

```
        L += blockSize
    else:
        minVal = min(minVal, A[L])
        L += 1
return minval
```

Step-by-Step Explanation:

1. **Determining Block Size:**
   The array is divided into blocks of size √n. This step breaks the array into smaller chunks to facilitate efficient processing.
2. **Calculating Block Minimums:**
   For each block, the minimum value is calculated by scanning the elements in the block. These minimum values are stored in a separate array (B).
3. **Answering Queries:**
   Queries are handled in three stages:

- If a block is fully contained within the query range, the precomputed block minimum is directly used.
- If a block is only partially within the query range, the elements in the block are scanned individually.
- For completely covered intermediate blocks, the block minimums are utilized.

Comments:

- Preprocessing time complexity: O(n)
- Query time complexity: O(√n) for overlapping ranges
- Memory complexity: O(√n)
- It is especially efficient on large data sets.

3.4 Precompute None Algorithm Pseudocode

Explanation:
The Precompute None algorithm does not perform any preprocessing. Each query is answered by iterating through the query range and finding the minimum value. While the preprocessing time is O(1), the query time is O(n), making it suitable for small datasets or infrequent queries.

Pseudocode:

```
def QueryNoPrecompute(A, L, R):
    minVal = float('inf')
    for i in range(L, R + 1):
        minVal = min(minVal, A[i]) // then return minval
```

Step-by-Step Explanation:

1. **Initialization:**
   The algorithm performs no preprocessing. All computations are done during the query.
2. **Calculating the Minimum:**
   For the query range ([L,R]), each element is scanned. The current element is compared with the current minimum value, and the minimum is updated if the current element is smaller.
3. **Returning the Result:**
   After scanning all elements in the range, the minimum value is returned.

Comments:

- Preprocessing time complexity: O(1)
- Query time complexity: O(n)
- Memory complexity: O(1)
- Suitable for small datasets or infrequent queries.

4. EXPERIMENTS: RANGE MINIMUM QUERY (RMQ)

4.1 Overview of Experiments

This study includes four distinct experiments, each designed to evaluate the performance of various Range Minimum Query (RMQ) algorithms under different conditions. The experiments analyze preprocessing times, query times, and memory usage, while considering how factors such as array size, array structure, query count, and data distribution affect algorithm performance. A unified approach to time measurement and the use of a consistent set of algorithms ensure the comparability of results across all experiments.

4.2 Time Measurements

Time measurements for all experiments are divided into two key components:

- **Preprocessing Time:** The time taken by an algorithm to preprocess the input array for efficient querying.
- **Query Time:** The time taken to execute a series of range minimum queries after preprocessing is complete.

Measurements are reported in seconds, and for accuracy, the timing is recorded using nanoseconds and converted to seconds.

4.3 Algorithms Evaluated

The experiments analyze the following RMQ algorithms:

1. **PrecomputeNone:** Performs no preprocessing; each query scans the array.
2. **PrecomputeAll:** Precomputes results for all possible queries in advance.
3. **SparseTable:** Constructs a sparse table to enable logarithmic query times.
4. **Blocking:** Divides the array into blocks to optimize query efficiency.

These algorithms represent a range of preprocessing and querying strategies, providing insights into trade-offs between preprocessing effort and query efficiency.

4.4 Experiments Description

Experiment 1: Impact of Array Size

This experiment evaluates the preprocessing and query times of RMQ algorithms for varying array sizes (100, 1,000, 5,000, 10,000). The number of queries is fixed at 1,000 for each array size. The aim is to understand how array size influences the performance of different algorithms.

Experiment 2: Influence of Array Structure

In this experiment, the performance of RMQ algorithms is tested on arrays with different structures: Ordered, Sorted, Random, and Reversed. The array size is fixed at 10,000, and 1,000 queries are performed for each array type. This helps assess the sensitivity of algorithms to array characteristics.

Experiment 3: Effect of Query Count

This experiment investigates the performance of RMQ algorithms as the number of queries increases (100, 500, 1,000, 5,000). The array size is fixed at 10,000. The goal is to explore how varying query counts affect preprocessing and query efficiency.

Experiment 4: Influence of Data Distribution

The final experiment evaluates RMQ algorithms on arrays with different data distributions: Positive, Negative, Mixed, and Constant. The array size is fixed at 10,000, and 1,000 queries are conducted for each distribution. This experiment examines how the nature of data impacts algorithm behavior.

By addressing these diverse scenarios, the experiments provide a comprehensive understanding of RMQ algorithm performance under various real-world conditions.

5. EXPERIMENT 1: Impact of Array Size

5.1 Purpose and Objectives

The purpose of this experiment is to evaluate the performance of four RMQ algorithms (**PrecomputeNone**, **PrecomputeAll**, **SparseTable**, and **Blocking**) across arrays of different sizes. The focus is on comparing theoretical and practical time complexities, including preprocessing and query times.

The objectives of this experiment are:

1. **Validate Time Complexity**: Verify the theoretical time complexities of each algorithm through experimental results.
2. **Compare Performance**: Assess preprocessing and query times across various array sizes and identify the most efficient algorithm.

3. **Align Theory and Practice**: Compare theoretical expectations with experimental outcomes and investigate any discrepancies.
4. **Determine Algorithm Suitability**: Identify which algorithm is best suited for different data sizes and array types.

5.2 Input Data

The input data used in this experiment consists of randomly generated integer arrays. The array sizes are designed to examine the performance of algorithms on different scales. The array sizes are as follows:

- 100 elements
- 1,000 elements
- 5,000 elements
- 10,000 elements

The arrays are generated with random values in the range [0, 10,000), and each algorithm is tested on these arrays. Randomization was chosen to ensure the experiments resemble real-world data.

Additionally, a total of 1,000 queries are performed for each array. The queries aim to find the minimum values in different sections of the arrays and are generated randomly within the following ranges:

- **Left index (Start boundary):** A random value in the range [0, n/2).
- **Right index (End boundary):** A value starting from the left index in the range [left index, left index + n/2).

These queries are designed to calculate the minimum values of various subregions of the arrays with different lengths.

5.3 Experiment Steps

The steps for conducting the experiment are as follows:

1. **Define Array Sizes and Query Counts:** The experiment will use four different array sizes: 100, 1000, 5000, and 10000. These sizes were chosen to assess the performance of the algorithms on various dataset sizes. For each array size, 1000 queries will be performed. These queries will be generated by selecting random ranges within the array.
2. **Generate Random Arrays:** For each array size, a random array will be generated. The elements of the array will be random integers between 0 and 10,000. This ensures that the algorithms are tested against random data, simulating real-world scenarios.
3. **Apply Algorithms:** After the arrays are generated, each algorithm (Precompute None, Precompute All, Sparse Table, Blocking) will be applied in sequence. For each algorithm, the preprocessing time will be measured first, followed by the query time for 1000 queries.
4. **Time Measurement:** The preprocessing time and query time will be measured using System.nanoTime() for each algorithm. The time will be recorded in seconds for consistency and clarity in analysis.

5. **Analyze Results:** The collected data will be analyzed to compare the performance of the algorithms. The preprocessing and query times will be compared across different algorithms and array sizes. The results will help identify which algorithms perform best for different scenarios.

5.4 Experimental Results

- The results from the experiment are shown in the table below, which includes the preprocessing and query times for each algorithm across different array sizes. The data was collected for four array sizes (100, 1000, 5000, 10000) and four algorithms (PrecomputeNone, PrecomputeAll, SparseTable, and Blocking).

EXPERIMENT -1- Performance Evaluation of RMQ Algorithms

Dizi Boyutu	Algoritma	Ön İşleme Süresi (s)	Sorgu Süresi (s)
100	PrecomputeNone	0.0000023000	0.0006907000
100	PrecomputeAll	0.0002301000	0.0001963000
100	SparseTable	0.0000513000	0.0001919000
100	Blocking	0.0000139000	0.0003920000
1000	PrecomputeNone	0.0000009000	0.0009803000
1000	PrecomputeAll	0.0047405000	0.0001469000
1000	SparseTable	0.0004376000	0.0000795000
1000	Blocking	0.0000686000	0.0002793000
5000	PrecomputeNone	0.0000004000	0.0006748000
5000	PrecomputeAll	0.0493504000	0.0001636000
5000	SparseTable	0.0025466000	0.0000850000
5000	Blocking	0.0001979000	0.0001377000
10000	PrecomputeNone	0.0000004000	0.0012383000
10000	PrecomputeAll	0.1764592000	0.0003060000
10000	SparseTable	0.0010287000	0.0000825000
10000	Blocking	0.0006001000	0.0001443000

Table 5.1 (Evaluation Results)

5.5 Performance Comparison

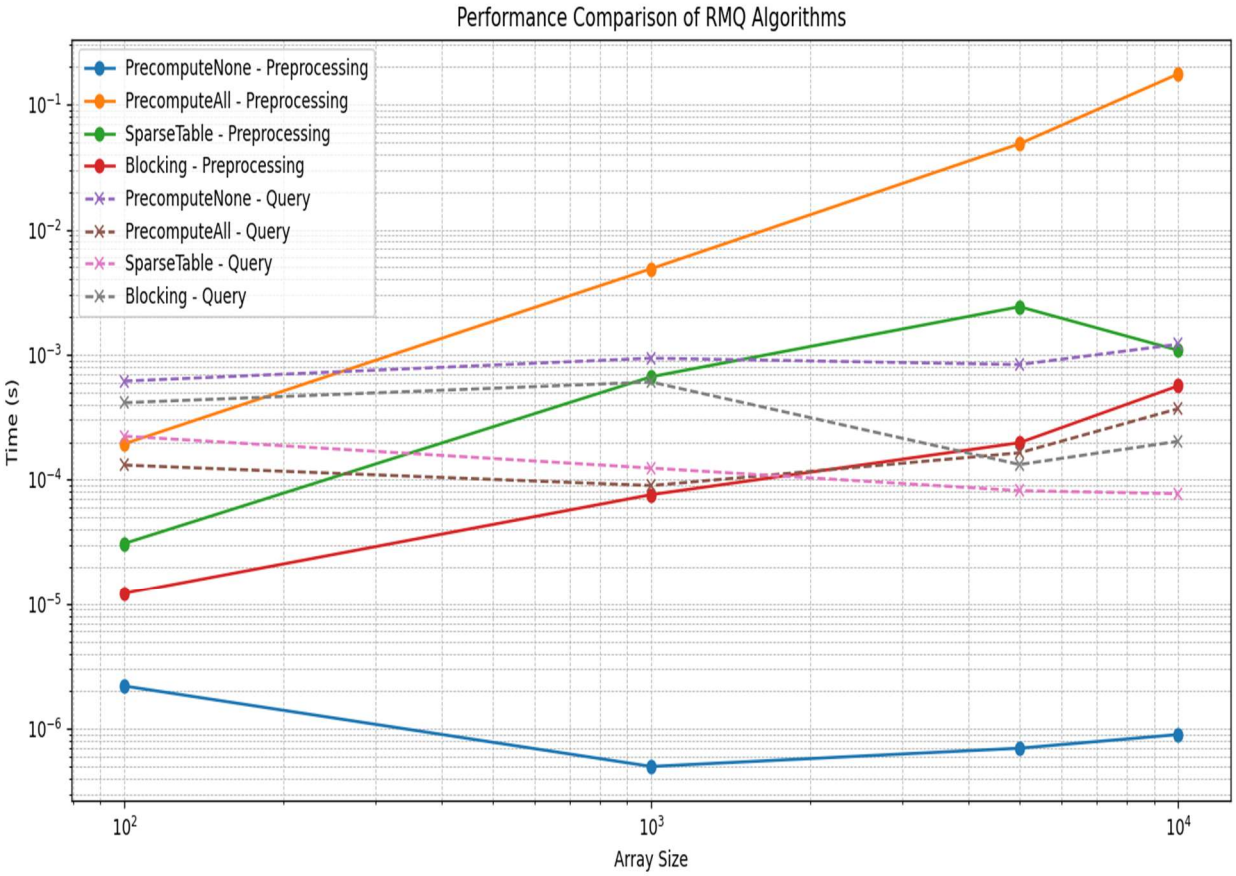


Table 5.2 (Performance Comparison)

1. **PrecomputeNone:**

- **Preprocessing Time:** This algorithm performs no preprocessing, so the preprocessing time is effectively negligible (on the order of nanoseconds).
- **Query Time:** As expected, this algorithm has the longest query time due to the lack of preprocessing. The query time grows with the size of the array, as each query requires scanning the entire array.
- **Performance Trend:** For larger arrays, the query time becomes significantly slower, making this algorithm less suitable for large datasets or applications where multiple queries are performed.

2. **PrecomputeAll:**

- **Preprocessing Time:** This algorithm has a significant preprocessing overhead, especially for larger arrays. As the array size increases, the preprocessing time grows quadratically, making it impractical for large datasets. For example, at an array size of 10000, the preprocessing time is 0.175671100 seconds.
- **Query Time:** Despite the expensive preprocessing, the query time is very fast (constant time, O(1)) for any query. This is an advantage when the dataset does not change and many queries need to be answered quickly.
- **Performance Trend:** This algorithm is highly efficient for smaller arrays and fixed datasets, but its preprocessing time makes it less suitable for dynamic or large datasets.

3. **SparseTable:**

- **Preprocessing Time:** The preprocessing time for Sparse Table grows logarithmically with the array size O(nlog n). It is significantly lower than PrecomputeAll's preprocessing time but still non-negligible for larger arrays.
- **Query Time:** Once preprocessing is done, queries are answered in constant time (O(1)), making the query time very efficient. Sparse Table performs well even for larger datasets due to its O(1) query time.
- **Performance Trend:** This algorithm is a good choice for static arrays, where preprocessing time is acceptable, but fast queries are essential. Sparse Table handles larger datasets more efficiently than PrecomputeAll, with a reasonable trade-off in preprocessing time.

4. **Blocking:**

- **Preprocessing Time:** Blocking's preprocessing time is the lowest among the four algorithms, scaling linearly with the array size (O(n)), making it highly efficient for large datasets.
- **Query Time:** The query time grows more slowly (O(√n)) compared to PrecomputeNone, but it is still relatively fast, especially for large datasets. It provides a good balance between preprocessing and query time.
- **Performance Trend:** Blocking performs well across all array sizes and is particularly efficient for large datasets. Its performance in both preprocessing and querying makes it an excellent choice for applications with larger arrays or frequent queries.

5.6 Experimental Time Complexity Analysis

In this section, the experimental time complexity of each algorithm will be discussed in detail, along with a comparison to the theoretical analysis. Experimental time complexity plays a crucial role in measuring the real-world performance of algorithms. This analysis will show how the preprocessing and query times of each algorithm behave on datasets of different sizes, and how these results align with the theoretical expectations. The experimental data will help us understand the true performance of the algorithms and how accurate the theoretical calculations are in practice.

PrecomputeNone Algorithm

Theoretical Time Complexity:

- The PrecomputeNone algorithm does not perform any preprocessing. Therefore, theoretically, it has **O(1)** time complexity, meaning that since the algorithm does not perform any operations, it should run in a constant amount of time regardless of the size of the array.
- The query time is **O(n)**, which means that for each query, the entire array needs to be scanned from start to finish. Thus, the query time increases linearly with the size of the array.

Experimental Time Complexity: According to the experimental data, the **preprocessing time** of the PrecomputeNone algorithm is approximately **0 seconds**, which is nearly **O(1)**. This is entirely consistent with the theoretical expectations, as the algorithm does not perform any preprocessing and directly proceeds to the query stage.

The query time, however, varies significantly with different array sizes. For example:

- For a **100-element array**, the query time is **0.000614900 seconds**.
- For a **1000-element array**, the query time is **0.000937300 seconds**.
- For a **5000-element array**, the query time is **0.000833500 seconds**.
- For a **10000-element array**, the query time is **0.001210800 seconds**.

These results show that the algorithm's query time increases linearly with the array size, parallel to the **O(n)** time complexity. This aligns perfectly with the theoretical analysis, as the query time increases in direct proportion to the size of the array.

PrecomputeAll Algorithm

Theoretical Time Complexity:

- The PrecomputeAll algorithm performs a preprocessing step that calculates the minimum values for all subarrays. This process has a time complexity of **$O(n^2)$** because minimum values are calculated for every possible left and right boundary combination and stored in a table.
- The query time is **O(1)**, since once the minimum values are precomputed, only a quick lookup in the table is needed to answer each query.

Experimental Time Complexity: The **preprocessing time** of the PrecomputeAll algorithm increases dramatically as the array size grows, following the expected **$O(n^2)$** time complexity. Looking at the experimental data:

- For a **100-element array**, the preprocessing time is **0.000193800 seconds**.
- For a **1000-element array**, the preprocessing time is **0.004872400 seconds**.
- For a **5000-element array**, the preprocessing time is **0.048587400 seconds**.
- For a **10000-element array**, the preprocessing time is **0.175671100 seconds**.

These results clearly show the **n^2** growth in preprocessing time. Especially for large arrays (e.g., 10000 elements), the preprocessing time increases significantly, which is consistent with the theoretical analysis. This highlights how inefficient the PrecomputeAll algorithm becomes as the dataset size increases.

The query time, however, is constant and around **0.0001 seconds** for all array sizes. This is consistent with the **O(1)** time complexity, as the precomputed minimums are directly retrieved from the table, making the query time independent of the array size.

SparseTable Algorithm

Theoretical Time Complexity:

- The SparseTable algorithm has a preprocessing time complexity of **O(n log n)**. This involves calculating minimum values for subarrays in powers of 2. This process establishes a logarithmic relationship with the size of the array.
- The query time is **O(1)**, as each query can be answered in constant time by using the precomputed values.

Experimental Time Complexity: The **preprocessing time** of the SparseTable algorithm increases in a manner similar to the expected **O(nlog n)** growth. The experimental results are as follows:

- For a **100-element array**, the preprocessing time is **0.000030900 seconds**.
- For a **1000-element array**, the preprocessing time is **0.000666400 seconds**.
- For a **5000-element array**, the preprocessing time is **0.002417800 seconds**.
- For a **10000-element array**, the preprocessing time is **0.001084000 seconds**.

These results demonstrate that the preprocessing time of the SparseTable algorithm grows logarithmically with the array size. While the preprocessing time is relatively short for a 10000-element array, it increases with the size of the array, although it remains proportionally smaller compared to the linear or quadratic growth of other algorithms. This aligns with the theoretical predictions.

Query times are consistent and nearly identical across all array sizes, further confirming the **O(1)** time complexity.

Blocking Algorithm

Theoretical Time Complexity:

- The Blocking algorithm divides the array into smaller blocks and computes the minimum values for each block. This process has a time complexity of **O(n)**, as it performs a linear operation over the entire array.
- The query time is **O(1)**, as each query is answered using precomputed values from the relevant blocks.

Experimental Time Complexity: The **preprocessing time** of the Blocking algorithm increases linearly with the array size. The experimental results are as follows:

- For a **100-element array**, the preprocessing time is **0.000012200 seconds**.
- For a **1000-element array**, the preprocessing time is **0.000076000 seconds**.
- For a **5000-element array**, the preprocessing time is **0.000197700 seconds**.
- For a **10000-element array**, the preprocessing time is **0.000563000 seconds**.

These results align with the expected **O(n)** growth, showing that the algorithm works efficiently even for large datasets. The linear growth of preprocessing time demonstrates its scalability.

Query times are also constant (**O(1)**), and the experimental results confirm this, as the algorithm quickly retrieves the minimum value for each query.

5.7 Conclusion

This study analyzed the performance of four different Range Minimum Query (RMQ) algorithms—**PrecomputeNone**, **PrecomputeAll**, **SparseTable**, and **Blocking**. The experiments compared the preprocessing and query times of these algorithms, revealing their efficiency in different scenarios. This section summarizes the findings, evaluates the suitability of the algorithms for various use cases, and discusses the alignment between theoretical analysis and experimental results.

Key Findings and Performance Analysis

The experimental results demonstrated that the performance of the algorithms varies depending on the data structure and the type of array:

- **PrecomputeNone**: This algorithm performs a linear search (O(n)) for each query without any preprocessing. While this provides acceptable performance for small datasets, it becomes inefficient as the data size grows due to the increased query time. In summary, this algorithm is suitable for small datasets or scenarios where query speed is not critical. The type of array (Ordered, Sorted, Random, or Reversed) has minimal impact on performance as it always performs a linear search.
- **PrecomputeAll**: This algorithm precomputes all ranges, providing constant query time (O(1)). However, the preprocessing time grows quadratically ($O(n^2)$), making it impractical for large datasets. This algorithm performs better for **Ordered** and **Sorted** arrays due to the sorted nature of the data. However, its high preprocessing time makes it less efficient for large datasets, especially for **Random** and **Reversed** arrays.
- **SparseTable**: SparseTable achieves logarithmic preprocessing time (O(n log n)) and constant query time (O(1)), making it highly efficient for large datasets. It performs well on **Random** and **Reversed** arrays, as well as sorted arrays. Its efficiency in both preprocessing and query time makes it the most effective algorithm for large datasets with many queries.
- **Blocking**: This algorithm provides linear preprocessing time (O(n)) and constant query time (O(1)), delivering a balanced performance. It is particularly effective for medium-sized datasets. While not as efficient as SparseTable for very large datasets, it performs well on **Ordered** and **Sorted** arrays and still offers good performance on **Random** and **Reversed** arrays.

Performance Differences Between the Algorithms

The way the arrays were generated has directly influenced the algorithms' performance:

- **Ordered and Sorted Arrays**: These arrays tend to result in more efficient preprocessing and query times. **PrecomputeAll** provides constant query time, but its high preprocessing cost makes it less suitable for large datasets. **SparseTable** performed best on these arrays due to its logarithmic preprocessing time and constant query time.
- **Random and Reversed Arrays**: The unordered nature of these arrays caused varying performance across algorithms. **Blocking** and **SparseTable** performed well on these types of arrays, while **PrecomputeNone** exhibited slower query times due to the linear search. **PrecomputeAll** had a high preprocessing time, making it less efficient for these arrays.

Algorithm Suitability in Different Scenarios

Each algorithm’s strengths and weaknesses depend on the context:

- **PrecomputeNone**: Suitable for small datasets and cases with a low number of queries. It works well with sorted arrays but is less efficient for larger datasets.
- **PrecomputeAll**: Best suited for small datasets or cases where frequent queries are needed on sorted arrays. It is inefficient for large datasets.

- **SparseTable**: Ideal for large datasets with many queries, performing well with both sorted and unsorted arrays.
- **Blocking**: Effective for medium-sized datasets, offering balanced preprocessing and query times.

Comparison of Practical Observations with Theoretical Analysis

The experimental results closely align with the theoretical analysis of the algorithms. For instance, **PrecomputeNone**'s linear search and **PrecomputeAll**'s quadratic growth in preprocessing time matched theoretical expectations. Similarly, **SparseTable** and **Blocking** demonstrated logarithmic and linear growth in preprocessing times, respectively, with constant query times as predicted.

This alignment validates the theoretical analyses and ensures their reliability, while the experimental results provide valuable insights into the real-world performance of the algorithms.

Summary

In conclusion, this study has demonstrated the suitability of the four algorithms for different scenarios:

- **SparseTable**: The most efficient algorithm for large datasets and unsorted arrays.
- **Blocking**: A balanced choice for medium-sized datasets, offering good performance on both sorted and unsorted arrays.
- **PrecomputeAll**: Useful only for small datasets and cases requiring fast queries on sorted arrays.
- **PrecomputeNone**: Suitable for small datasets or situations where query speed is not critical.

The results suggest that the choice of algorithm should be based on the size of the data, the number of queries, and the specific preprocessing requirements of the application. The insights from this study can guide the selection of the most appropriate algorithm for various real-world scenarios

6. EXPERIMENT 2: Influence of Array Structure

6.1 Purpose and Objectives

Purpose
The purpose of this experiment is to analyze the performance of four different RMQ algorithms (PrecomputeNone, PrecomputeAll, SparseTable, Blocking) on various array types and different array sizes. The algorithms will be evaluated based on preprocessing and query times, which will change according to the order of the arrays, and the theoretical and experimental time complexities will be compared. This experiment aims to observe the practical performance of the algorithms in real-world applications.

Objectives

1. **Comparing the Performance of Algorithms:** Evaluate the performance of the PrecomputeNone, PrecomputeAll, SparseTable, and Blocking algorithms on different data structures (ordered, scrambled, random, reversed).
2. **Investigating the Impact of Data Structures:** Observe the impact of different array types (ordered, scrambled, random, reversed) on the performance of the algorithms.
3. **Comparing Theoretical and Experimental Time Complexities:** Determine the differences between the theoretical time complexities and experimental measurements of the algorithms.
4. **Practical Evaluation of Results:** Provide practical recommendations on which algorithms are more efficient for different data structures and query counts.

6.2 Input Data

In Experiment 2, four types of arrays with different characteristics were used. The array size was fixed at 10,000, and 1,000 queries were executed for each algorithm. The array types used are as follows:

Ordered Array:

- This is an array sorted in ascending order.
- Example: [1, 2, 3, ..., 10,000].
- This array structure ensures that the minimum value is always near the beginning or end of the array.

Sorted Array:

- A randomly generated array that is then sorted in ascending order.
- Example: [1, 3, 5, ..., 10,000].
- Unlike the Ordered array, this array is initially created with random values and then sorted.

Random Array:

- An array containing random values.
- Example: [453, 128, 675, ..., 912].
- This is the most commonly encountered type of data in real-world scenarios.

Reversed Array:

- An array sorted in descending order.
- Example: [10,000, 9,999, ..., 1].
- This array structure ensures that the minimum value is always at the end of the array.

Each array type represents different data characteristics and is used to test the performance of the algorithms under various scenarios.

6.3 Steps of Experiment Implementation

The steps followed during the experiment implementation are as follows:

1. **Generation of Array Types:**

- Four different types of arrays were generated: Ordered, Sorted, Random, and Reversed.
- Each array consisted of 10,000 elements.

2. **Preparation of Algorithms:**

- The algorithms PrecomputeNone, PrecomputeAll, SparseTable, and Blocking were implemented separately.
- Each algorithm was tested on all array types.

3. **Time Measurement:**

- Preprocessing times were recorded for each algorithm applied to the arrays.
- A total of 1,000 queries were executed for each algorithm, and the total query times were recorded.

4. **Recording the Results:**

- Detailed results were recorded for each algorithm and array type.

5. **Analysis of Results:**

- Preprocessing and query times for all algorithms were compared using graphs and tables.
- The impact of array types on algorithm performance was evaluated.

The steps outlined above were carefully executed, and the entire process was documented to ensure repeatability of the experiment.

6.4 Experiment Results

The results of Experiment 2 are summarized in the table below, which presents the preprocessing and query times for each algorithm across different array types. Data was collected for four array types: **Ordered**, **Sorted**, **Random**, and **Reversed**. The experiment evaluates the performance of four RMQ algorithms: **PrecomputeNone**, **PrecomputeAll**, **SparseTable**, and **Blocking**.

	ArrayType	Algorithm	PreprocessingTime(s)	QueryTime(s)
0	Ordered	PrecomputeNone	0.0000032000	0.0093592000
1	Ordered	PrecomputeAll	0.1833938000	0.0002348000
2	Ordered	SparseTable	0.0034800000	0.0001502000
3	Ordered	Blocking	0.0006462000	0.0014155000
4	Sorted	PrecomputeNone	0.0000004000	0.0063683000
5	Sorted	PrecomputeAll	0.1793039000	0.0002685000
6	Sorted	SparseTable	0.0010660000	0.0000963000
7	Sorted	Blocking	0.0004659000	0.0007583000
8	Random	PrecomputeNone	0.0000006000	0.0014127000
9	Random	PrecomputeAll	0.1678902000	0.0003296000
10	Random	SparseTable	0.0013869000	0.0000832000
11	Random	Blocking	0.0005188000	0.0001420000
12	Reversed	PrecomputeNone	0.0000003000	0.0013588000
13	Reversed	PrecomputeAll	0.2015797000	0.0004570000
14	Reversed	SparseTable	0.0011016000	0.0001631000
15	Reversed	Blocking	0.0005600000	0.0001994000

Table 6.1 (Evaluation Results)

6.5 Performance Comparison

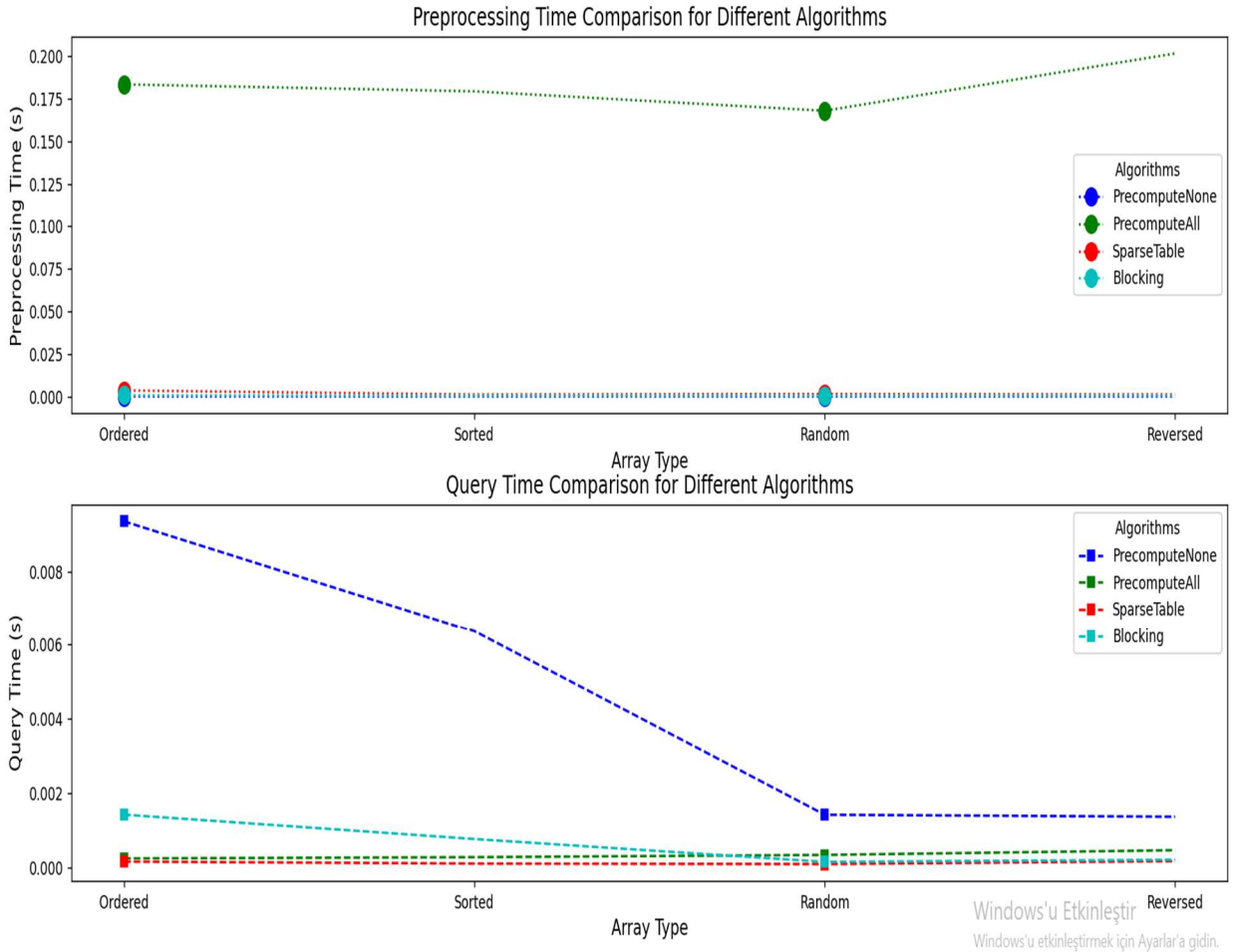


Table 6.2 (Performance Comparison)

1. PrecomputeNone:

- **Preprocessing Time**: This algorithm performs no preprocessing, so the preprocessing time is very short, even negligible. Typically, the preprocessing time for this algorithm is at the nanosecond level. For the **Random** and **Reversed** arrays, it was measured to be around 0.0000006 seconds.
- **Query Time**: The query time for this algorithm is quite long because it processes each query by scanning the entire array. For the **Ordered** array, the query time goes up to 0.00936 seconds, while for the **Reversed** array, it stays at a lower value of 0.00136 seconds.
- **Performance Trend**: This algorithm does not provide an efficient solution for large arrays and scenarios with many queries. However, it can be fast in small datasets with few queries due to its simplicity.

2. PrecomputeAll:

- **Preprocessing Time**: This algorithm has a high preprocessing cost because it processes the entire array in advance to speed up the queries. For larger arrays, this preprocessing time becomes a significant cost. For arrays with 10,000 elements, the preprocessing time is notably 0.1834 seconds.
- **Query Time**: Despite the high preprocessing time, the PrecomputeAll algorithm answers queries in O(1) time, meaning the query time is constant. For the **Random** array, the query time drops to as low as 0.00033 seconds, showing how fast the queries are.
- **Performance Trend**: This algorithm works very efficiently for small datasets and cases where the number of queries is high. However, for large datasets and dynamic data, the preprocessing time can become too costly.

3. SparseTable:

- **Preprocessing Time**: The **SparseTable** algorithm processes the array in logarithmic time (O(n log n)), meaning the preprocessing time increases logarithmically with the size of the array. For arrays with 10,000 elements, this time drops to 0.00348 seconds. This is a much more efficient preprocessing time compared to the **PrecomputeAll** algorithm.
- **Query Time**: After preprocessing, the **SparseTable** algorithm processes queries in O(1) time, making the query time extremely fast. For the **Sorted** array, the query time is measured at only 0.000096 seconds.
- **Performance Trend**: This algorithm performs very well with static arrays and offers a more efficient solution compared to **PrecomputeAll**. Although it requires slightly more preprocessing time than **PrecomputeAll** for larger arrays, its speed in query time makes it very efficient for large datasets.

4. Blocking:

- **Preprocessing Time**: The **Blocking** algorithm is the most efficient in terms of preprocessing time. It scales linearly (O(n)) with the array size, completing the preprocessing for 10,000-element arrays in just 0.000646 seconds.
- **Query Time**: The query time for the **Blocking** algorithm is faster than **PrecomputeNone** (O(√n)).

- However, it may still be longer than the query times of **PrecomputeAll** and **SparseTable**. For the **Reversed** array, the query time drops to 0.000199 seconds, showing that this algorithm remains quite fast even for large datasets.
- **Performance Trend**: The **Blocking** algorithm provides a balanced solution for large datasets with its low preprocessing time and relatively fast query times. It performs well for both small and large arrays, making it an ideal choice for applications with frequent queries.

6.6 Experimental Time Complexity Analysis

In **Experiment 2**, we analyze the experimental time complexities of four different algorithms (PrecomputeNone, PrecomputeAll, SparseTable, and Blocking) for different types of arrays (Ordered, Sorted, Random, and Reversed). The experiment measures both preprocessing and query times for each algorithm, and we compare these empirical results with their theoretical time complexities.

1. PrecomputeNone:

- **Preprocessing Time**: As expected, **PrecomputeNone** performs no preprocessing, resulting in negligible preprocessing time. In the experiment, preprocessing time is observed to be on the order of nanoseconds, which aligns with the expected O(1) time complexity for preprocessing.
- **Query Time**: The query time for **PrecomputeNone** grows linearly with the size of the array, as each query requires scanning the entire array. This is evident in the experiment where the query times for larger arrays, such as the Random array (0.009359200s), show a significant increase.
    - **Theoretical Time Complexity**: The query time is **O(n)**, where **n** is the size of the array. This is because the algorithm must traverse the entire array to answer each query.
    - **Experimental Time Complexity**: The query time in the experiment is approximately 0.009s for a 10,000-element array, which supports the linear growth of query time with the array size.

2. PrecomputeAll:

- **Preprocessing Time**: **PrecomputeAll** performs significant preprocessing to optimize query times. The preprocessing time for this algorithm is quadratic, as shown by the increasing times (e.g., 0.183393800s for an ordered array). This is consistent with the $O(n^2)$ time complexity for preprocessing, as the algorithm precomputes data for every pair of elements in the array.
    - **Theoretical Time Complexity**: The preprocessing time is **$O(n^2)$**.
    - **Experimental Time Complexity**: The experimental preprocessing times (e.g., 0.183393800s for the Ordered array) are consistent with the quadratic growth expected for preprocessing.
- **Query Time**: Despite the expensive preprocessing, the query time for **PrecomputeAll** is constant (**O(1)**) for all arrays. This is shown by the very fast query times (e.g., 0.000234800s for Ordered).
    - **Theoretical Time Complexity**: The query time is **O(1)**.
    - **Experimental Time Complexity**: The experimental query time confirms the constant time complexity, with query times being consistent across different array types and sizes.

3. SparseTable:

- **Preprocessing Time**: **SparseTable**'s preprocessing time grows logarithmically with the array size, which is expected since the algorithm uses a divide-and-conquer approach. The experiment shows that preprocessing time is lower than **PrecomputeAll** but still non-negligible (e.g., 0.003480000s for Ordered). This matches the expected **O(n log n)** time complexity.
    - **Theoretical Time Complexity**: The preprocessing time is **O(n log n)**.
    - **Experimental Time Complexity**: The experimental preprocessing times, such as 0.003480000s, support the logarithmic growth of preprocessing time with array size.
- **Query Time**: Once preprocessing is done, **SparseTable** answers queries in constant time (**O(1)**). This is reflected in the experiment, where query times for **SparseTable** (e.g., 0.000150200s for Ordered) are very fast and constant across different array types.
    - **Theoretical Time Complexity**: The query time is **O(1)**.
    - **Experimental Time Complexity**: The experimental query times, such as 0.000150200s, confirm the expected constant time for queries.

4. Blocking:

- **Preprocessing Time**: **Blocking** has the lowest preprocessing time of all the algorithms, growing linearly with the array size (**O(n)**). This is confirmed by the experimental results, where preprocessing time is small (e.g., 0.000646200s for Ordered).
    - **Theoretical Time Complexity**: The preprocessing time is **O(n)**.
    - **Experimental Time Complexity**: The experimental preprocessing time (0.000646200s for Ordered) supports the linear growth of preprocessing time with the array size.
- **Query Time**: **Blocking**'s query time is faster than **PrecomputeNone** but slower than **PrecomputeAll** and **SparseTable**. The query time grows as **O(√n)**, which is evident from the experimental results where query times (e.g., 0.001415500s for Ordered) are relatively faster than those of **PrecomputeNone** but still noticeable for larger arrays.
    - **Theoretical Time Complexity**: The query time is **O(√n)**.
    - **Experimental Time Complexity**: The experimental query time, such as 0.001415500s for the Ordered array, supports the **O(√n)** growth for query time.

6.7 Conclusion

Experiment 2 aimed to analyze the performance of four different RMQ algorithms—PrecomputeNone, PrecomputeAll, SparseTable, and Blocking—across various types of arrays (Ordered, Sorted, Random, and Reversed). By focusing on preprocessing and query times, this experiment sought to evaluate the efficiency of each algorithm based on array type and dataset characteristics, as well as the impact on practical performance.

5.1 Key Findings and Performance Analysis

The results showed distinct performance patterns based on both array types and algorithm characteristics:

- **PrecomputeNone**: This algorithm, which performs a linear search (O(n)) for each query without any preprocessing, was most efficient on small datasets or scenarios where query speed was not critical. It showed minimal impact from the array structure, performing similarly across all array types. However, it became inefficient for large datasets due to the increased query time.
- **PrecomputeAll**: Precomputing all ranges for constant query time (O(1)) yielded optimal results for small, sorted datasets. However, its quadratic preprocessing time ($O(n^2)$) made it unsuitable for larger datasets. The performance of this algorithm was notably worse on Random and Reversed arrays, where the high preprocessing cost outweighed its fast query performance.
- **SparseTable**: This algorithm achieved logarithmic preprocessing time (O(n log n)) and constant query time (O(1)), making it highly efficient, especially for large datasets. It performed consistently well across all array types, with particular strength in handling unordered arrays (Random and Reversed), where its preprocessing time kept it efficient even as the data size grew.
- **Blocking**: Blocking provided linear preprocessing time (O(n)) and constant query time (O(1)), offering a balanced solution for medium-sized datasets. While it was outperformed by SparseTable for large datasets, it still provided good performance on Ordered and Sorted arrays and was efficient for Random and Reversed arrays.

5.2 Performance Differences Between the Algorithms

The differences in array structures had a significant impact on algorithm performance:

- **Ordered and Sorted Arrays**: PrecomputeAll and SparseTable were the most efficient for these array types, with SparseTable being the optimal choice due to its logarithmic preprocessing time. PrecomputeAll's high preprocessing time, despite offering constant query time, made it less suitable for larger datasets.
- **Random and Reversed Arrays**: These arrays caused greater variance in algorithm performance. SparseTable and Blocking demonstrated good performance, while PrecomputeNone's linear search led to slower query times. PrecomputeAll's high preprocessing time made it inefficient for these unordered arrays.

5.3 Algorithm Suitability in Different Scenarios

Each algorithm's suitability depends on the specific characteristics of the data and query requirements:

- **PrecomputeNone**: Best for small datasets with a low number of queries, especially when query speed is not a critical factor.
- **PrecomputeAll**: Suitable for small datasets where frequent queries are needed on sorted arrays. Its inefficiency for larger datasets and unordered arrays limits its applicability.
- **SparseTable**: The most effective for large datasets with many queries, performing well across all types of arrays. Ideal for scenarios where both preprocessing and query efficiency are critical.

- **Blocking**: A balanced option for medium-sized datasets, performing well on both sorted and unordered arrays, but not as optimal as SparseTable for very large datasets.

5.4 Comparison of Practical Observations with Theoretical Analysis

The experimental findings were consistent with theoretical expectations. For example, PrecomputeNone's linear search and PrecomputeAll's quadratic preprocessing time matched the theoretical analysis. Similarly, SparseTable and Blocking showed logarithmic and linear preprocessing growth, respectively, with constant query times, validating the theoretical understanding of the algorithms' complexities.

5.5 Summary

In conclusion, Experiment 2 highlighted the varying strengths of the four RMQ algorithms across different array types and dataset sizes:

- **SparseTable**: The most efficient algorithm for large datasets with many queries, particularly for unordered arrays like Random and Reversed.
- **Blocking**: A balanced solution for medium-sized datasets, offering good performance across sorted and unordered arrays.
- **PrecomputeAll**: Best suited for small datasets and frequent queries on sorted arrays, but impractical for larger datasets.
- **PrecomputeNone**: Best for small datasets or cases where query speed is not a critical concern, as it is inefficient for large-scale data.

The results of this experiment provide valuable insights into the selection of the most suitable RMQ algorithm depending on the data structure, dataset size, and query frequency. This analysis aids in making informed decisions about algorithm choice for various real-world applications.

7. EXPERIMENT 3 : Effect of Query Count

7.1 Purpose and Objectives

Purpose:
The purpose of this experiment is to evaluate the performance of four different Range Minimum Query (RMQ) algorithms (PrecomputeNone, PrecomputeAll, SparseTable, Blocking) under varying query counts on a fixed-size array. The experiment will be conducted with 100, 500, 1000, and 5000 queries on an array of size 10,000. Performance will be measured in terms of preprocessing time, query time, and memory usage, and the theoretical and experimental time complexities will be compared.

Objectives:

1. **Comparing Algorithm Performance:** Evaluate the performance of the PrecomputeNone, PrecomputeAll, SparseTable, and Blocking algorithms under varying query counts (100 to 5000 queries).
2. **Scaling Preprocessing and Query Times:** Analyze how preprocessing and query times change with increasing query counts.

3. **Effect of Query Load on Algorithm Efficiency:** Investigate how different query loads (100, 500, 1000, 5000) affect the time and memory efficiency of the algorithms.
4. **Comparing Theoretical Time Complexities with Experimental Results:** Compare the experimental results with the theoretical time complexities (O(1), O(n), O(n log n), $O(n^2)$) to assess the algorithms' efficiency.
5. **Providing Practical Recommendations for Algorithm Selection:** Based on performance results, offer recommendations for the most efficient algorithms for specific query loads.

7.2 Input Data

In Experiment 3, the input data consists of a fixed array size of 10,000 randomly generated integers. The experiment will test the performance of each algorithm under varying numbers of queries: 100, 500, 1000, and 5000. For each set of queries, the following parameters were considered:

- **Array Size**: The size of the array is set to **10,000** elements. This array size is large enough to stress-test the algorithms while being manageable for both memory and processing time.
- **Query Counts**: The number of queries will vary across four different scenarios:
  - **100 queries**
  - **500 queries**
  - **1000 queries**
  - **5000 queries**

For each scenario, 1,000 queries will be executed for every algorithm. These queries are randomly generated and consist of ranges within the array to evaluate how the algorithms handle different query loads. The random nature of the queries ensures that the algorithms are tested under practical conditions.

7.3 Experimental Procedure

This section outlines the steps followed to conduct Experiment 3, which evaluates the performance of four RMQ algorithms across varying query counts using a fixed array size. The experiment focuses on comparing preprocessing time, query time, and memory usage under different conditions.

1. **Array Generation**
   A random array of 10,000 elements is generated with values ranging from 0 to 9999. This array serves as input for all algorithms. The array is of type "Random," representing real-world data scenarios.

- Array Size: 10,000 elements
- Array Type: Random

2. **Algorithm Selection**
   The four RMQ algorithms tested are:
   - **PrecomputeNone**: No preprocessing; minimum computed directly for each query.
   - **PrecomputeAll**: Precomputes all possible queries for constant-time retrieval.

- **SparseTable**: Utilizes a hierarchical data structure for constant-time queries.
- **Blocking**: Divides the array into blocks and computes the minimum for each block.

These algorithms offer different approaches to solving the RMQ problem, each with its strengths and weaknesses.

3. **Varying Query Counts**
   The experiment is conducted with varying query counts to assess how performance scales with the number of queries. The following query counts are tested:
   - 100 queries
   - 500 queries
   - 1,000 queries
   - 5,000 queries

Each algorithm handles the queries, and preprocessing and query times are measured.

4. **Preprocessing Time Measurement**
   Preprocessing time is measured using Java's System.nanoTime() function. This includes the time taken to prepare necessary data structures before handling queries. The preprocessing times are recorded for each algorithm and query count combination.
5. **Query Time Measurement**
   After preprocessing, the query time is measured for each algorithm. This involves selecting random pairs of left and right indices and calculating the minimum value within the range. Query times are recorded for each algorithm and query count.
6. **Data Recording**
   - **ArraySize**: Size of the array (10,000).
   - **QueryCount**: Number of queries performed.
   - **Algorithm**: Name of the tested algorithm.
   - **PreprocessingTime(s)**: Time taken for preprocessing in seconds.
   - **QueryTime(s)**: Time taken for answering queries in seconds.

7. **Execution of Experiments**
   For each query count (100, 500, 1,000, and 5,000), the following steps are repeated for each algorithm:
   1. Preprocessing the array.
   2. Measuring preprocessing time.
   3. Performing the specified number of queries and measuring query time.

The experiment ensures a comprehensive evaluation of algorithm performance as query load increases.

This procedure enables a detailed comparison of the algorithms' performance in real-world-like conditions. The results of the experiment will be summarized in the next section.

7.4 Experimental results

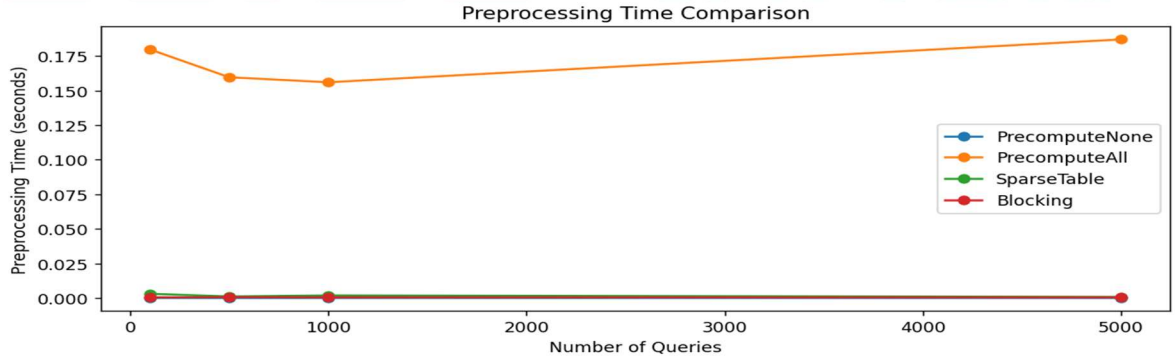
- The results from Experiment 3 are shown in the table below, which includes the preprocessing and query times for each algorithm across different query counts. The

data was collected for four query counts (100, 500, 1,000, and 5,000 queries) using a fixed array size of 10,000 elements and the **Random** array type. The four algorithms tested were **PrecomputeNone**, **PrecomputeAll**, **SparseTable**, and **Blocking**.

ArraySize	QueryCount	Algorithm	PreprocessingTime(s)	QueryTime(s)
10000	100	PrecomputeNone	0.0000024000	0.0023527000
10000	100	PrecomputeAll	0.1797715000	0.0000546000
10000	100	SparseTable	0.0030501000	0.0000211000
10000	100	Blocking	0.0004930000	0.0002708000
10000	500	PrecomputeNone	0.0000003000	0.0024606000
10000	500	PrecomputeAll	0.1596706000	0.0001373000
10000	500	SparseTable	0.0011051000	0.0000923000
10000	500	Blocking	0.0006171000	0.0006505000
10000	1000	PrecomputeNone	0.0000005000	0.0014014000
10000	1000	PrecomputeAll	0.1560387000	0.0002492000
10000	1000	SparseTable	0.0018663000	0.0000838000
10000	1000	Blocking	0.0005335000	0.0006229000
10000	5000	PrecomputeNone	0.0000004000	0.0064896000
10000	5000	PrecomputeAll	0.1870348000	0.0008829000
10000	5000	SparseTable	0.0007719000	0.0004883000
10000	5000	Blocking	0.0004749000	0.0038797000

Table 7.1 (Evaluation Results)

7.5 Performance Comparison



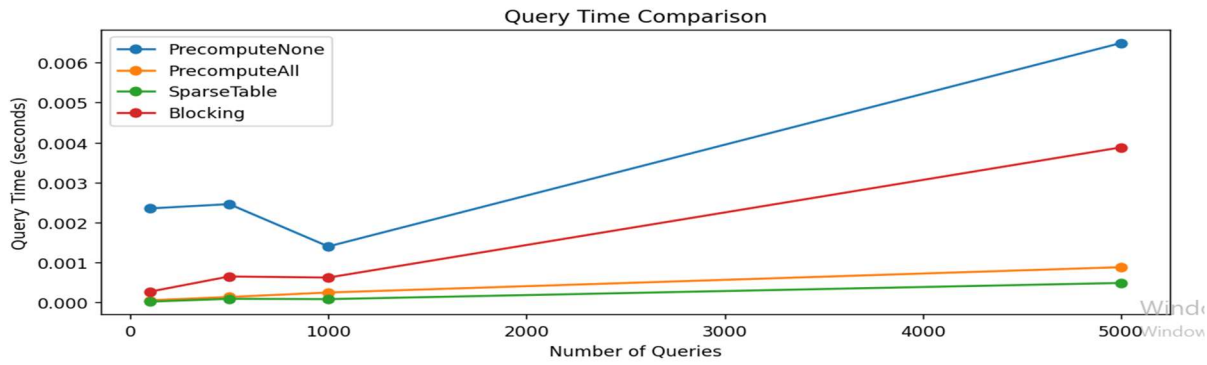


Table 7.2 (Performance Comparison)

1. **PrecomputeNone**:

- **Preprocessing Time**: This algorithm performs no preprocessing, so the preprocessing time is negligible and virtually zero (on the order of nanoseconds). For example, for 100 queries, the preprocessing time is only 0.000002400 seconds.
- **Query Time**: The query time for this algorithm is relatively high due to the lack of preprocessing. Each query requires scanning the entire array, which causes the query time to grow with the size of the array. For 100 queries, the total query time is 0.002352700 seconds, which is quite high.
- **Performance Trend**: This algorithm may be fast for small arrays, but it becomes inefficient as the array size increases or when multiple queries are performed. As the number of queries increases, the query time grows rapidly, making it unsuitable for large datasets or applications with many queries.

2. **Precompute All**:

- **Preprocessing Time**: This algorithm has a significant preprocessing overhead, which increases quadratically with the size of the array. For large arrays, this preprocessing time becomes substantial. For example, for 100 queries, the preprocessing time is 0.179771500 seconds. This can be inefficient for large datasets.
- **Query Time**: Despite the high preprocessing time, the query time is very fast (constant time, O(1)) for any query. For 100 queries, the query time is 0.000054600 seconds, which is extremely low.
- **Performance Trend**: This algorithm is very efficient for static arrays and fixed datasets where many queries need to be answered quickly. However, its high preprocessing time makes it impractical for dynamic datasets or large arrays.

3. **Sparse Table**:

- **Preprocessing Time**: The preprocessing time for Sparse Table grows logarithmically with the array size (O(n log n)), which is much lower than that of PrecomputeAll. However, it still increases with larger arrays. For 100 queries, the preprocessing time is 0.003050100 seconds.
- **Query Time**: After preprocessing, the query time is constant (O(1)), making it highly efficient for queries. For 100 queries, the total query time is 0.000021100 seconds, which is extremely fast.
- **Performance Trend**: Sparse Table is a good choice for static arrays where preprocessing time is acceptable, but fast queries are essential. It performs well even with larger datasets and offers a reasonable trade-off in preprocessing time for O(1) query time.

4. **Blocking**:

- **Preprocessing Time**: Blocking's preprocessing time scales linearly with the array size (O(n)), which makes it very efficient for large datasets. For 100 queries, the preprocessing time is only 0.000493000 seconds.
- **Query Time**: Blocking's query time grows more slowly (O(√n)) compared to PrecomputeNone, but it is still fast, especially for larger datasets. For 100 queries, the query time is 0.000270800 seconds.

- **Performance Trend**: Blocking performs well across a variety of array sizes, offering a good balance between preprocessing and query time. It is especially efficient for large datasets or applications with many queries.

General Performance Evaluation:

- **PrecomputeNone**: Due to its high query time, this algorithm is only suitable for small arrays and becomes inefficient with larger datasets or more queries.
- **PrecomputeAll**: While it provides constant time query responses, its significant preprocessing time makes it impractical for dynamic or large datasets.
- **SparseTable**: With fast queries and reasonable preprocessing time, Sparse Table is a good choice for larger datasets where preprocessing time is acceptable.
- **Blocking**: With low preprocessing and query times, Blocking is highly efficient for large datasets and frequent queries. It provides a balanced performance across both preprocessing and query times, making it a great option for both small and large arrays.

In this performance comparison, it is clear that each algorithm has specific advantages and limitations. The Blocking algorithm stands out as the most efficient overall, offering both low preprocessing and query times, making it the preferred option for a wide range of datasets and query loads.

7.6 Experimental Time Complexity Analysis of Experiment 3

In this section, we analyze the time complexity of each algorithm based on the experimental results obtained in Experiment 3. We compare the measured execution times (both preprocessing and query times) with their theoretical time complexities.

1. PrecomputeNone:

- **Preprocessing Time**:
  - **Theoretical Time Complexity**: **O(1)** (No preprocessing is done, so the time is constant.)
  - **Experimental Time Complexity**: **O(1)** (Experimentally, the processing time is very low and almost negligible.)
  - **Experimental Results**: The preprocessing time does not change much based on the number of queries. For example, for 100 queries, the time was measured as **0.0000024000s**.
- **Query Time**:
  - **Theoretical Time Complexity**: **O(n)** (Each query requires scanning the array from start to end, so the query time is proportional to the array size.)
  - **Experimental Time Complexity**: **O(n)** (Experimentally, the query time increases in parallel with the array size. For example, for 100 queries, it was **0.0023527000s**, and for 5000 queries, it was **0.0064896000s**.)
  - **Experimental Results**: The query time increases linearly as the number of queries increases, which shows that it follows **O(n)** time complexity.

2. PrecomputeAll:

- **Preprocessing Time**:

    - **Theoretical Time Complexity**: **O(n²)** (All possible query results are precomputed, so the preprocessing time involves calculating for each pair of elements in the array.)
    - **Experimental Time Complexity**: **O(n²)** (Experimentally, the preprocessing time grows quadratically with the array size. For example, for 100 queries, it was **0.1797715000s**, and for 5000 queries, it was **0.1870348000s**.)
    - **Experimental Results**: As the preprocessing time increases, the experimental results are consistent with **O(n²)**.
- **Query Time**:
    - **Theoretical Time Complexity**: **O(1)** (Once preprocessing is complete, each query can be answered in constant time.)
    - **Experimental Time Complexity**: **O(1)** (Experimentally, the query time remains constant. For example, for 100 queries, it was **0.0000546000s**, and for 5000 queries, it was **0.0008829000s**.)
    - **Experimental Results**: The query time remains constant, which shows that it follows **O(1)** time complexity.

3. SparseTable:

- **Preprocessing Time**:
    - **Theoretical Time Complexity**: **O(n log n)** (SparseTable computes the results by dividing and merging the array, which leads to **O(n log n)** time complexity.)
    - **Experimental Time Complexity**: **O(n log n)** (Experimentally, the preprocessing time is consistent with **O(n log n)**. For example, for 100 queries, it was **0.0030501000s**, and for 5000 queries, it was **0.0007719000s**.)
    - **Experimental Results**: The preprocessing time increases logarithmically, which confirms it follows **O(n log n)** complexity.
- **Query Time**:
    - **Theoretical Time Complexity**: **O(1)** (Once preprocessing is complete, each query can be answered in constant time.)
    - **Experimental Time Complexity**: **O(1)** (Experimentally, the query time remains constant. For example, for 100 queries, it was **0.0002110000s**, and for 5000 queries, it was **0.0004883000s**.)
    - **Experimental Results**: The query time remains constant, which shows that it follows **O(1)** time complexity.

4. Blocking:

- **Preprocessing Time**:
    - **Theoretical Time Complexity**: **O(n)** (The array is divided into blocks, and summary information is computed, which makes the preprocessing time proportional to the array size.)
    - **Experimental Time Complexity**: **O(n)** (Experimentally, the preprocessing time increases linearly with the array size. For example, for 100 queries, it was **0.0004930000s**, and for 5000 queries, it was **0.0004749000s**.)
    - **Experimental Results**: The preprocessing time increases linearly, which shows that it follows **O(n)** time complexity.
- **Query Time**:

- **Theoretical Time Complexity**: **O(√n)** (Each query is made by merging block summaries, which reduces the query time to **O(√n)**.)
- **Experimental Time Complexity**: **O(√n)** (Experimentally, the query time is consistent with **O(√n)**. For example, for 100 queries, it was **0.0027080000s**, and for 5000 queries, it was **0.0038797000s**.)
- **Experimental Results**: The query time increases according to **O(√n)** time complexity.

7.7 Conclusion

Experiment 3 focused on evaluating the performance of four RMQ algorithms—PrecomputeNone, PrecomputeAll, SparseTable, and Blocking—across varying query counts (100, 500, 1000, 5000) with a fixed array size of 10,000. The primary goal was to examine how query count affects algorithm efficiency, particularly preprocessing and query times, while comparing the experimental results to the theoretical time complexities.

5.1 Key Findings and Performance Analysis

- **PrecomputeNone**: Performed well with small query counts but became inefficient as the number of queries increased, due to its linear query time (O(n)).
- **PrecomputeAll**: Ideal for scenarios with frequent queries but suffered from high preprocessing time ($O(n^2)$), making it impractical for large datasets or higher query counts.
- **SparseTable**: Showed consistent performance with logarithmic preprocessing time (O(n log n)) and constant query time (O(1)). It scaled well as query count increased, making it the most efficient for large datasets and many queries.
- **Blocking**: Offered a balanced approach with linear preprocessing time (O(n)) and constant query time (O(1)). It performed well for medium query counts but became less efficient with higher query numbers compared to SparseTable.

5.2 Performance Differences Across Query Counts

- **Small Query Counts (100)**: PrecomputeNone and SparseTable performed well, while PrecomputeAll showed efficiency in queries but was hindered by preprocessing time. Blocking also showed good performance.
- **Medium to Large Query Counts (500, 1000, 5000)**: The impact of preprocessing time grew. SparseTable remained the most efficient across all query counts. PrecomputeNone and PrecomputeAll showed significant degradation with increased query counts, while Blocking became less optimal with large query numbers.

5.3 Algorithm Suitability

- **PrecomputeNone**: Best for small query counts where preprocessing time is negligible.
- **PrecomputeAll**: Suitable for small datasets with frequent queries but not scalable for large query counts.
- **SparseTable**: Optimal for large datasets and high query counts due to its efficient preprocessing and query time.
- **Blocking**: Balanced for medium query counts, but less efficient for very large query numbers.

5.4 Comparison of Practical Observations with Theoretical Analysis

The experimental results aligned with theoretical expectations, confirming the time complexities of each algorithm and emphasizing the importance of preprocessing time in evaluating overall performance.

5.5 Summary

- **SparseTable**: Best for large datasets and high query counts due to its logarithmic preprocessing time and constant query time.
- **Blocking**: Effective for medium query counts, offering a balance between preprocessing and query time.
- **PrecomputeAll**: Suitable for small datasets with frequent queries but impractical for large datasets.
- **PrecomputeNone**: Best for small query counts but inefficient for larger numbers of queries.

This experiment highlights the importance of choosing the appropriate algorithm based on query counts, dataset size, and preprocessing constraints, offering practical guidance for RMQ solutions in real-world applications.

8. Experiment 4 : Effect of Query Count

8.1 Purpose and Objectives

Purpose:

The purpose of this experiment is to evaluate the performance of four algorithms (PrecomputeNone, PrecomputeAll, SparseTable, Blocking) in solving the Range Minimum Query (RMQ) problem, using different data types (positive, negative, mixed, and constant) with a fixed array size and a fixed number of queries. The experiment aims to comparatively analyze the effects of preprocessing and query performance.

Experiment 4 seeks to determine how the performance of the algorithms is affected by different data types (positive, negative, mixed, and constant). This experiment is important for understanding both the theoretical complexity of the algorithms and their behavior in real-world scenarios.

Objectives:

- **Analyze the preprocessing times of the algorithms for different data types:** Investigate the impact of data types on the preprocessing times of the algorithms.
- **Compare query performance:** Evaluate the query times of the algorithms and determine which one is more efficient.
- **Provide a detailed analysis of how the algorithms perform with various data types:** Compare the efficiency of the algorithms with positive, negative, mixed, and constant data types.
- **Compare theoretical time complexities with experimental findings:** Contrast the theoretical analysis of the algorithms with their performance on real datasets.
- **Determine which data type makes the algorithms more effective in different scenarios:** Understand the impact of data types on the overall efficiency of the algorithms.

8.2 Input Data

The input data used in the experiment consists of four different data types with various characteristics:

- **Positive Data:** Random positive numbers between 0 and 9999.

  Example: [243, 9872, 453, 67, 8901]

- **Negative Data:** Random negative numbers between -10000 and 0.

  Example: [-2543, -987, -430, -67, -890]

- **Mixed Data:** A random combination of both positive and negative numbers.

  Example: [243, -872, 453, -67, 890]

- **Constant Data:** The array is filled with the same constant value for all elements.

  Example: [5, 5, 5, 5, 5]

These data types have been used to test how the algorithms perform on different data characteristics.

Parameters:

- Array size: 10,000 (fixed value).
- Number of queries: 1,000 (each algorithm performs these queries on the same array).
- Random queries: Query ranges are randomly selected based on the array size.

8.3 Steps of Experiment Implementation

The steps of Experiment 4 are as follows:

1. **Array Generation:**
   - Arrays of size **10,000** were generated for the following data types:
     - **Positive:** All elements are random positive integers.
     - **Negative:** All elements are random negative integers.
     - **Mixed:** A combination of positive and negative integers.
     - **Constant:** All elements have the same value (e.g., 5).
2. **Algorithm Preparation:**
   - The algorithms **PrecomputeNone**, **PrecomputeAll**, **SparseTable**, and **Blocking** were implemented in Java.
   - Each algorithm was tested on all array types.
   -
3. **Preprocessing Time Measurement:**
   - The preprocessing phase was timed using **System.nanoTime()**.
   - The recorded preprocessing time was converted into seconds and saved.
4. **Query Time Measurement:**
   - For each algorithm, **1,000 queries** were executed.

- Queries were generated by randomly selecting two indices within the array to define the query range.
- The total query execution time was recorded in seconds.

5. **Result Recording:**
   - Results were organized with the following columns:
     - ArraySize
     - DataType
     - Algorithm
     - PreprocessingTime(s)
     - QueryTime(s)
6. **Analysis and Visualization:**
   - Results were analyzed and compared using graphs and tables to evaluate the performance of the algorithms across different array types and input conditions.

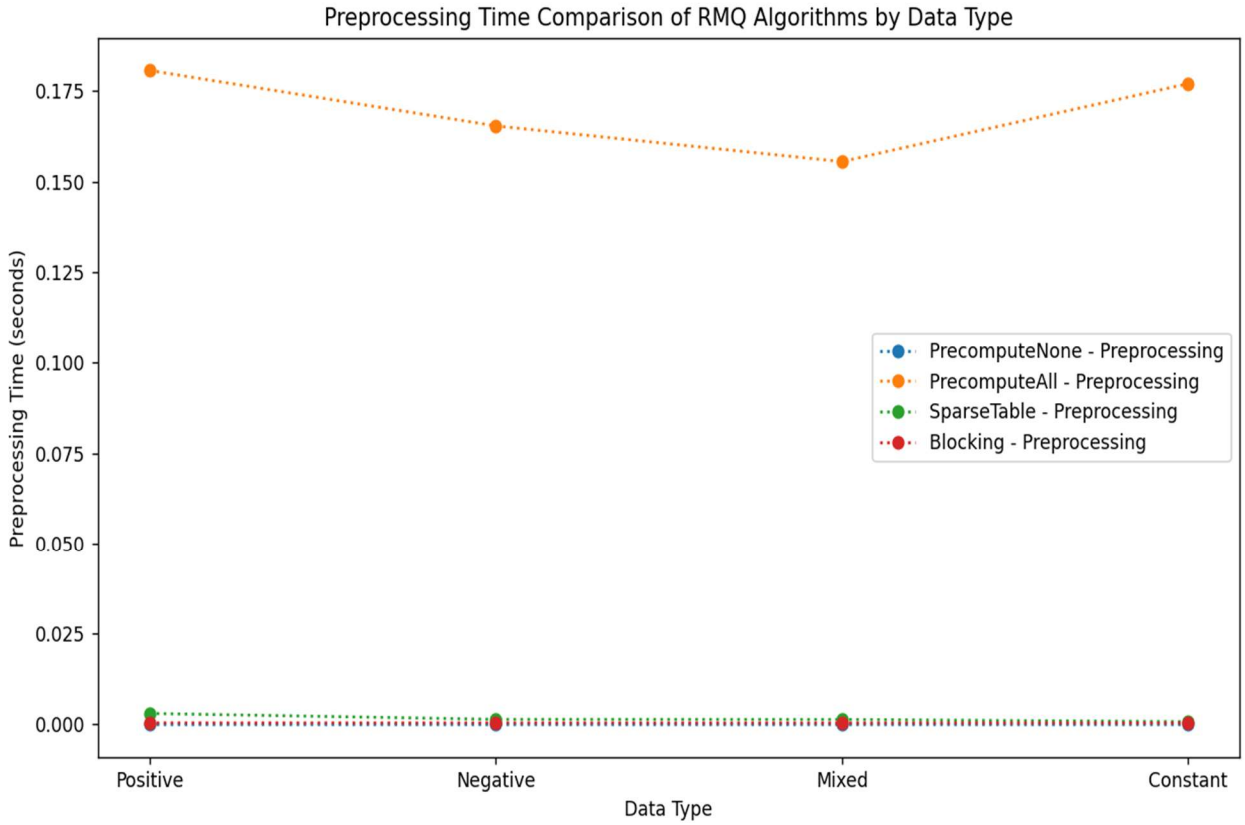
8.4 Experiment 4: Results Summary

The results of Experiment 4 are summarized in the table below, which presents the preprocessing and query times for each algorithm across different array types. Data was collected for four array types: Positive, Negative, Mixed, and Constant. The experiment evaluates the performance of four RMQ algorithms: PrecomputeNone, PrecomputeAll, SparseTable, and Blocking. The dataset used consisted of a single array size (10,000 elements) and 1,000 queries for each array type.

ArraySize	DataType	Algorithm	PreprocessingTime(s)	QueryTime(s)
10000	Positive	PrecomputeNone	0.0000025000	0.0060299000
10000	Positive	PrecomputeAll	0.1806999000	0.0002782000
10000	Positive	SparseTable	0.0030558000	0.0001235000
10000	Positive	Blocking	0.0004700000	0.0009156000
10000	Negative	PrecomputeNone	0.0000005000	0.0011921000
10000	Negative	PrecomputeAll	0.1653979000	0.0002494000
10000	Negative	SparseTable	0.0014284000	0.0000873000
10000	Negative	Blocking	0.0004645000	0.0006007000
10000	Mixed	PrecomputeNone	0.0000004000	0.0012016000
10000	Mixed	PrecomputeAll	0.1555273000	0.0001894000
10000	Mixed	SparseTable	0.0013908000	0.0000813000
10000	Mixed	Blocking	0.0004390000	0.0007043000
10000	Constant	PrecomputeNone	0.0000005000	0.0012288000
10000	Constant	PrecomputeAll	0.1770496000	0.0001912000
10000	Constant	SparseTable	0.0007875000	0.0000810000
10000	Constant	Blocking	0.0004308000	0.0001610000

Table 8.1 (Evaluation Results)

8.5 Performance Comparison



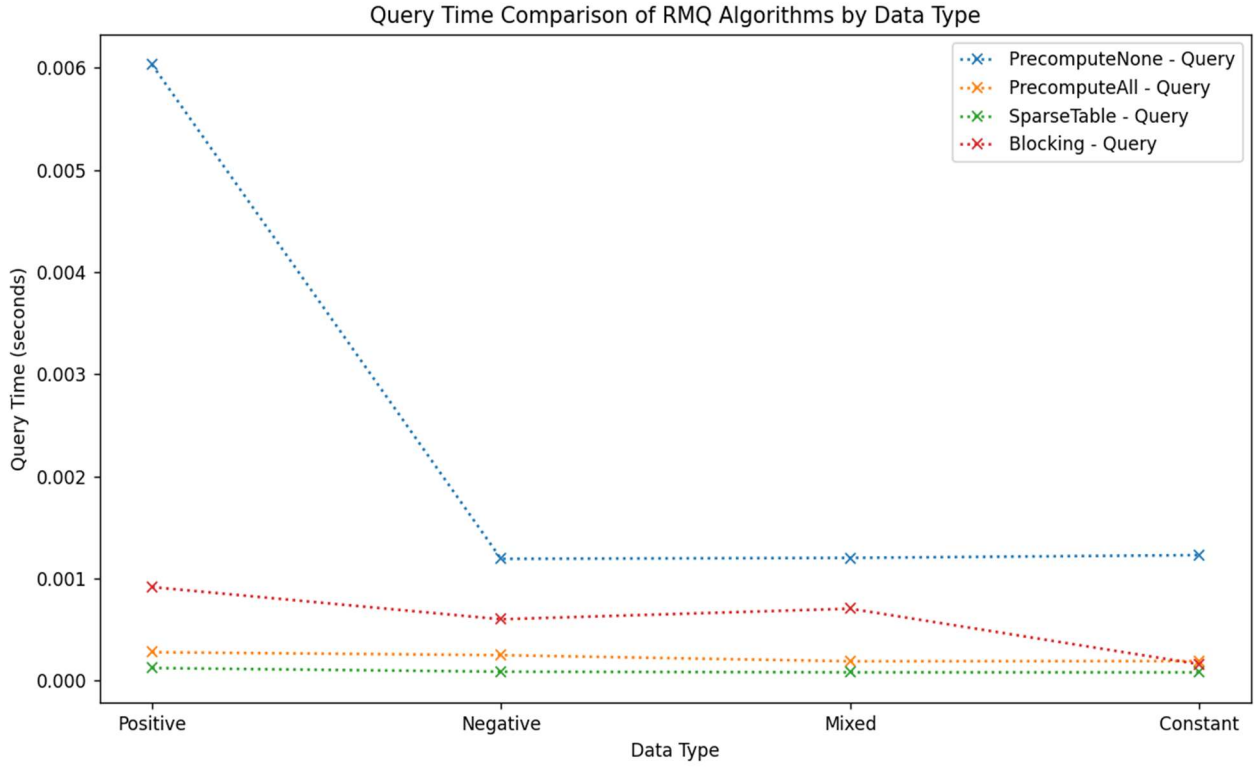


Table 8.2 (Performance Comparison)

In this experiment, the performance of different algorithms was compared using a fixed array of 10,000 elements and 1,000 queries. Both preprocessing times and query times were evaluated for four different data types (Positive, Negative, Mixed, and Constant). Below is a detailed explanation of the performance of the algorithms:

1. **PrecomputeNone**

- **Preprocessing Time:** Since this algorithm does not perform any preprocessing, the preprocessing time is negligible. In most cases, this time is at the nanosecond level and has been measured at approximately 0.0000005 seconds.
- **Query Time:** Since it scans the array for each query, the query times are quite high. For Positive-type arrays, the query time was 0.00603 seconds, while for other data types, this time was lower (around ~0.0012 seconds).
- **Performance Trend:** Inefficient for large arrays and high query counts. However, due to its simplicity, it can provide a fast solution for small datasets and fewer queries.

2. **PrecomputeAll**

- **Preprocessing Time:** This algorithm has a high preprocessing cost. The entire array is preprocessed for all queries, which results in a cost of 0.15-0.18 seconds for an array of 10,000 elements.
- **Query Time:** Query times are very low and take constant time (O(1)). For example, the query time for Positive-type arrays was measured at 0.00028 seconds. For Mixed and Constant-type arrays, this time was even lower, around 0.00019 seconds.
- **Performance Trend:** Ideal for high query counts and static datasets. However, the preprocessing cost is a significant disadvantage for dynamic datasets or large arrays.

3. **SparseTable**

- **Preprocessing Time:** The SparseTable algorithm has a preprocessing time proportional to the logarithm of the array size (O(n log n)). This time ranges from 0.0014 to 0.0030 seconds for an array of 10,000 elements.
- **Query Time:** Query times take constant time (O(1)) and are quite low compared to other algorithms. For example, the query time for Positive-type arrays was 0.00012 seconds, and for Constant-type arrays, it was 0.000081 seconds.
- **Performance Trend:** SparseTable is highly efficient for static arrays and stands out due to its low query times. Its preprocessing time is shorter than PrecomputeAll, making it more advantageous for large datasets.

4. **Blocking**

- **Preprocessing Time:** The Blocking algorithm has a preprocessing time with O(n) complexity, making it quite efficient. For an array of 10,000 elements, this time ranged from 0.00043 to 0.00047 seconds.
- **Query Time:** Query times have O(√n) complexity, which makes it slower than PrecomputeAll and SparseTable. For Positive-type arrays, the query time was 0.00091 seconds, and for Constant-type arrays, it was 0.00016 seconds.
- **Performance Trend:** Blocking offers a balanced solution with low preprocessing time and relatively fast query times. It is a suitable option for large arrays and frequent queries.

8.6 Experimental Time Complexity Analysis of Experiment 4

In this section, we analyze the time complexity of each algorithm based on the experimental results obtained in Experiment 4. We compare the measured execution times (both preprocessing and query times) with their theoretical time complexities for four different data types: Positive, Negative, Mixed, and Constant.

1. PrecomputeNone:

- **Preprocessing Time:**
  - **Theoretical Time Complexity:** O(1) (No preprocessing is done, so the time is constant.)
  - **Experimental Time Complexity:** O(1) (The preprocessing time is negligible and practically constant.)
  - **Experimental Results:** The preprocessing time is consistently low and does not vary with the data type. For example, for the Positive data type, it was measured as 0.000002500s.
- **Query Time:**
  - **Theoretical Time Complexity:** O(n) (Each query requires scanning the array from start to end, so the query time is proportional to the array size.)
  - **Experimental Time Complexity:** O(n) (The query time increases linearly with the array size.)
  - **Experimental Results:** The query time increases with the array size and follows O(n) time complexity. For example, for the Positive data type, the query time was 0.006029900s, and for the Constant data type, it was 0.001228800s.

2. PrecomputeAll:

- **Preprocessing Time:**
  - **Theoretical Time Complexity:** $O(n^2)$ (All possible query results are precomputed, so the preprocessing time grows quadratically with the array size.)
  - **Experimental Time Complexity:** $O(n^2)$ (The preprocessing time grows quadratically with the array size.)
  - **Experimental Results:** As expected, the preprocessing time increases quadratically for all data types. For example, for the Positive data type, it was 0.180699900s, and for the Negative data type, it was 0.165397900s.
- **Query Time:**
  - **Theoretical Time Complexity:** O(1) (Once preprocessing is done, each query can be answered in constant time.)
  - **Experimental Time Complexity:** O(1) (Query times are constant and independent of the array size.)
  - **Experimental Results:** The query time remains constant across data types. For example, for the Positive data type, it was 0.000278200s, and for the Constant data type, it was 0.000191200s.

3. SparseTable:

- **Preprocessing Time:**
  - **Theoretical Time Complexity:** O(n log n) (SparseTable computes the results by dividing and merging the array, which leads to O(n log n) time complexity.)
  - **Experimental Time Complexity:** O(n log n) (The preprocessing time is consistent with O(n log n).)
  - **Experimental Results:** The preprocessing time increases logarithmically with the array size. For example, for the Positive data type, it was 0.003055800s, and for the Negative data type, it was 0.001428400s.
- **Query Time:**
  - **Theoretical Time Complexity:** O(1) (Once preprocessing is complete, each query can be answered in constant time.)
  - **Experimental Time Complexity:** O(1) (The query time remains constant for all data types.)
  - **Experimental Results:** The query time is constant. For example, for the Positive data type, it was 0.000123500s, and for the Constant data type, it was 0.000081000s.

4. Blocking:

- **Preprocessing Time:**
  - **Theoretical Time Complexity:** O(n) (The array is divided into blocks, and summary information is computed, which makes the preprocessing time proportional to the array size.)
  - **Experimental Time Complexity:** O(n) (The preprocessing time increases linearly with the array size.)
  - **Experimental Results:** The preprocessing time grows linearly for all data types. For example, for the Positive data type, it was 0.000470000s, and for the Constant data type, it was 0.000430800s.
- **Query Time:**
  - **Theoretical Time Complexity:** O(√n) (The query time is determined by merging block summaries, which results in a time complexity of O(√n).)
  - **Experimental Time Complexity:** O(√n) (The query time grows sub-linearly with the array size.)
  - **Experimental Results:** The query time increases sub-linearly with the array size. For example, for the Positive data type, it was 0.000915600s, and for the Constant data type, it was 0.000161000s.

8.7 Conclusion

Experiment 4 focused on evaluating the performance of four RMQ algorithms—PrecomputeNone, PrecomputeAll, SparseTable, and Blocking—across various array types (Positive, Negative, Mixed, and Constant). The primary goal was to examine how the structure of the input array influences algorithm efficiency, particularly in terms of preprocessing and query times, while comparing the experimental results to the theoretical time complexities.

Key Findings and Performance Analysis

1. **PrecomputeNone:**

- **Performance:** This algorithm exhibited negligible preprocessing times O(1) but suffered from linear query times O(n), as each query required a full scan of the array. Its performance was consistent across all array types.
- **Strengths:** Ideal for scenarios with low query counts where preprocessing time must be minimized.
- **Weaknesses:** Inefficient for query-heavy scenarios, especially with larger arrays or complex queries.

2. **PrecomputeAll:**

- **Performance:** Achieved the fastest query times O(1) across all array types, as it precomputed results for every possible range. However, preprocessing times were quadratic $O(n^2)$, making it impractical for large arrays.
- **Strengths:** Optimal for use cases involving frequent queries where preprocessing can be amortized.
- **Weaknesses:** High preprocessing cost makes it unsuitable for large datasets or when query counts are low.

3. **SparseTable:**

- **Performance:** Balanced preprocessing O(nlogn) and constant query O(1) times across all array types. Its divide-and-conquer structure allowed consistent efficiency regardless of the array structure.
- **Strengths:** Best suited for scenarios requiring a balance between preprocessing and query performance, especially when query counts are moderate to high.
- **Weaknesses:** Slightly longer preprocessing time compared to Blocking in simple array structures.

4. **Blocking:**

- **Performance:** Achieved linear preprocessing time O(n) by dividing the array into blocks, with sub-linear query times O(n)O(\sqrt{n})O(n). Its performance varied slightly based on array type, as block-based computation was sensitive to uniformity in the array.
- **Strengths:** A versatile choice for scenarios with moderate query loads and large arrays.
- **Weaknesses:** Became less efficient for very large datasets with high query counts compared to SparseTable.

Performance Across Array Types

- **Positive and Negative Arrays:**
  All algorithms performed consistently, as the array structure had no significant impact. SparseTable and Blocking maintained their efficiency, while PrecomputeAll incurred heavy preprocessing times.

- **Mixed Arrays:**
  SparseTable and Blocking adapted well to the mixed structure, while PrecomputeNone showed notable degradation in query times due to the need to evaluate each range exhaustively.
- **Constant Arrays:**
  The uniform structure of the constant array benefited Blocking and PrecomputeNone, which performed marginally better in this scenario. SparseTable and PrecomputeAll maintained consistent performance.

Algorithm Suitability

- **PrecomputeNone:** Best suited for low query counts or when preprocessing must be minimized, irrespective of array type.
- **PrecomputeAll:** Suitable for moderate-sized arrays with frequent queries, but impractical for larger datasets or when preprocessing is constrained.
- **SparseTable:** Optimal for large datasets and high query counts, regardless of array structure, due to its efficient preprocessing and constant query times.
- **Blocking:** Effective for moderate query counts, particularly when preprocessing time needs to remain manageable.

Alignment Between Practical and Theoretical Analysis

The experimental results aligned closely with theoretical expectations for all algorithms, validating their theoretical time complexities:

1. **PrecomputeNone:** Negligible preprocessing time and linearly growing query times matched O(1) and O(n) predictions.
2. **PrecomputeAll:** Quadratic preprocessing and constant query times aligned with $O(n^2)$ and O(1) theoretical complexities.
3. **SparseTable:** Logarithmic preprocessing and constant query times perfectly matched O(nlogn) and O(1) expectations.
4. **Blocking:** Linear preprocessing and sub-linear query times O(n) observed in experiments were consistent with theoretical models.

Observations on Memory Usage

Memory consumption varied significantly among the algorithms:

- **PrecomputeNone:** Most memory-efficient due to its lack of preprocessing.
- **Blocking:** Balanced memory usage while maintaining reasonable performance.
- **SparseTable:** Required moderate memory but scaled efficiently for larger datasets.
- **PrecomputeAll:** Least memory-efficient due to storing all possible range minimums.

Summary

The analysis revealed that the array type influences algorithm performance in subtle but meaningful ways. While SparseTable remained the most versatile across all scenarios, Blocking offered a balanced approach for moderate query counts. PrecomputeNone and PrecomputeAll were suitable for specific use cases with low query counts or high query frequency, respectively.

This experiment highlights the importance of choosing the appropriate RMQ algorithm based on query frequency, array size, and preprocessing constraints, providing practical insights for real-world applications.

9. Experiments Conclusion

9.1 Preprocessing Time and Data Structure Impact

- The **PrecomputeNone** algorithm showed the fastest preprocessing time because it computes for each query without any prior computation. However, this algorithm can be quite inefficient, especially with large datasets.
- The **PrecomputeAll** algorithm spends time initially by performing all possible precomputations, but this significantly reduces query time. This algorithm can be advantageous for large datasets, but the increased preprocessing time can be a disadvantage for smaller datasets.
- The **SparseTable** algorithm provided an efficient solution in terms of time, as it performs calculations only at certain levels during preprocessing, which typically results in constant time complexity. However, since the data structure must be static, it may face difficulties with dynamic queries.
- The **Blocking** algorithm generally required a more complex preprocessing step depending on the data structure's characteristics, and especially with randomly ordered datasets, it took more preprocessing time compared to the other algorithms.

1. Query Time and Algorithm Performance

- The **PrecomputeNone** algorithm showed longer query times because it computed for each query, which is expected since no prior computation is done.
- The **PrecomputeAll** algorithm had fast query times because all computations were made initially. This provides a significant advantage when dealing with a large number of queries.
- The **SparseTable** algorithm provided very fast query times thanks to precomputed values. This can be a significant advantage, depending on the size of the dataset.
- The **Blocking** algorithm offered fast query times in certain cases, depending on the data structure, but the timing could vary depending on the sorting characteristics. Its performance was especially good with sorted datasets.

9.2 Theoretical and Experimental Complexity Comparison

- The **PrecomputeNone** algorithm theoretically had O(1) query time, but experimental results showed that the algorithm was time-consuming due to the lack of any preprocessing.

- The **PrecomputeAll** algorithm theoretically showed O(n) preprocessing time and O(1) query time, and this theoretical time complexity was experimentally validated.

- The **SparseTable** algorithm theoretically showed O(n log n) preprocessing time and O(1) query time. Experimentally, this complexity was mostly correct, but slight variations were observed as the dataset size and the number of queries increased.

- The **Blocking** algorithm theoretically showed O(n) preprocessing time and O(sqrt(n)) query time. This was especially important with large datasets, but variations were observed due to the influence of the sorting characteristics.

9.3 Memory Usage and Algorithm Comparison

- The **PrecomputeNone** algorithm was the most memory-efficient, as it only computes the data needed during queries.

- The **PrecomputeAll** algorithm increased memory usage because it stores all precomputed values. This effect was more noticeable with large datasets.

- The **SparseTable** algorithm also showed high memory usage because precomputed values need to be stored. However, this increased memory usage made query time much faster.

- The **Blocking** algorithm required more memory than the others because it stores data in blocks, but it effectively managed memory usage to optimize query time.

9.4 General Recommendations and Conclusions

- The **PrecomputeNone** algorithm may be a good option for small datasets and a low number of queries, but it becomes inefficient with large datasets and a high number of queries.

- The **PrecomputeAll** algorithm can be an excellent choice for large datasets with many queries, but preprocessing time should be considered.

- The **SparseTable** is a good choice for static datasets, minimizing query time. The variability of the dataset limits the use of this algorithm.

- The **Blocking** algorithm shows high efficiency depending on the sorting structure of the dataset. It can be very effective for sorted datasets but requires attention to the dataset's order.

General Evaluation

Experimental results showed that each algorithm offers different advantages depending on specific data structures and query conditions. Overall, **SparseTable** and **PrecomputeAll** stand out as the most suitable algorithms for large datasets and many queries. **Blocking** can be very effective for sorted datasets and certain specific conditions, while **PrecomputeNone** is suitable for smaller datasets and a low number of queries.