FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

Student:

Name: Kevser

Surname: Arslan

ID Number:2121221030

Department: Computer Engineering

Project:

RMQ PROBLEM

Course:

Name: Algorithm

Introduction

This project focuses on the Range Minimum Queries (RMQ) problem. RMQ can be defined as the task of querying the minimum element within a specific range in an array. **For instance**, finding the smallest value between two indices in an array is a common problem that holds significant importance in various fields, particularly in data analysis, graph algorithms, and dynamic programming. RMQ serves as a fundamental building block in the resolution of more complex problems by utilizing efficient data structures and algorithms. The primary aim of this project is to investigate and implement various algorithms used for solving the RMQ problem. Four main algorithms will be examined: Precompute All, Sparse Table, Blocking, and Precompute None. A theoretical performance analysis will be conducted for each algorithm, and the results will be empirically tested. Additionally, the time complexity and performance of each algorithm will be compared, supported by findings presented in graphs and tables. This study aims to explore both the theoretical and practical aspects of the RMQ problem, identifying the most suitable solution methods through a comparative analysis of the algorithms. Ultimately, the project seeks to provide a deeper understanding of RMQ solutions.

Problem definition:

RMQ (Range Minimum Query) is the problem of finding the smallest element in a specific subarray (i..j) of an array.

Formal Definition: Given an array array[1..n] and two indices i, j ($1 \leq i \leq j \leq n$), an RMQ(i, j) query returns the minimum value in the subarray array[i..j].

Algorithm definition:

Precompute All: This algorithm precomputes the minimum values of all possible subsequences of the given array and stores them in memory. In this way, when the minimum value of any subsequence is asked, the answer is given very quickly by reading this value directly from memory. However, the disadvantage of this approach is that both a lot of processing power and memory space are needed to calculate and store the minimum values of all possible subsequences.

Sparse Table: The Sparse Table is a powerful technique for accelerating range minimum queries. By storing precomputed minimum values for strategically chosen subranges, it achieves remarkably fast query times, typically in constant time (O(1)).

Blocking: In the Blocking algorithm, the array is divided into a set of equally sized blocks. The minimum value within each block is calculated and stored beforehand. During a query, the algorithm efficiently determines the minimum value within the requested range by considering the minimum values of the blocks that intersect with the query range.

Precompute None: This algorithm lacks any precomputation steps. Consequently, each query requires a full scan of the relevant portion of the array, leading to potentially high time complexity for frequent queries.

pseudo-code of the algorithms

1)PrecomputeAll

```
class PrecomputeAll:
    initialize table as a 2D array

    constructor(data):
        preprocess(data)

    function preprocess(data):
        n = length of data
        create a 2D array table of size n x n

        for i from 0 to n-1:
            table[i][i] = data[i] // Single element range
            for j from i + 1 to n-1:
                table[i][j] = min(table[i][j-1], data[j])

    function query(left, right):
        return table[left][right]
```

2)SparseTable

```
class SparseTable:
    Variables:
        table  // 2D array for minimum values
        log    // 1D array for logarithmic values

    Method Preprocess(data):
        n = length of data
        log[1] = 0
        for i = 2 to n:
            log[i] = log[i / 2] + 1

        k = log[n] + 1
        table[0][i] = data[i], tüm i için
        for j = 1 to k - 1:
            for i = 0 to n - (1 << j):
                table[j][i] = min(table[j-1][i], table[j-1][i + (1 << (j-1))])
Metot Query(left, right):
    j = log[right - left + 1]
    min_value = min(table[j][left], table[j][right - (1 << j) + 1])
    döndür min_value
```

3)Blocking

```
CLASS Blocking:
    DATA: int[] data
    DATA: int[] blockMins
    DATA: int blockSize

    METHOD constructor(data: int[]):
        SET this.data = data
        CALL preprocess()

    METHOD preprocess():
        n = LENGTH of data
        // Determine block size as the ceiling of the square root of n
        blockSize = CEILING of (sqrt(n))
        numBlocks = CEILING of (n / blockSize)

     // Create array to store minimums of each block
     blockMins = NEW int[numBlocks]
     FILL blockMins WITH Integer.MAX_VALUE

     // Compute minimum for each block
     FOR i FROM 0 TO n-1:
         blockIndex = i / blockSize
         blockMins[blockIndex] = MIN(blockMins[blockIndex], data[i])

 METHOD query(left: int, right: int) RETURNS int:
     min = Integer.MAX_VALUE

     // Determine starting and ending blocks
     startBlock = left / blockSize
     endBlock = right / blockSize
         IF startBlock == endBlock THEN:
             // If both indices are in the same block
             FOR i FROM left TO right:
                 min = MIN(min, data[i])
         ELSE:
             // Process remaining elements in the start block
             FOR i FROM left TO (startBlock + 1) * blockSize - 1:
                 min = MIN(min, data[i])

             // Process full blocks in between
             FOR i FROM startBlock + 1 TO endBlock - 1:
                 min = MIN(min, blockMins[i])

             // Process elements in the end block
             FOR i FROM endBlock * blockSize TO right:
                 min = MIN(min, data[i])


         RETURN min
```

4) PrecomputeNone

```
FUNCTION rangeMinimumQuery(arr, left, right)
    DECLARE min AS arr[left]
    FOR i FROM left TO right DO
        IF arr[i] < min THEN
            min = arr[i]
        END IF
    END FOR
    RETURN min
END FUNCTION
```

Experimental Design

I would like to start by explaining the Experimental Design class for better understanding:

The ExperimentalDesign class is designed to evaluate and compare the performance of different algorithms under varying conditions. It accomplishes this by running two types of analysis: Scalability Analysis and Data Pattern Analysis. Each analysis focuses on different aspects of the algorithms' efficiency, providing a comprehensive evaluation framework.

The goal of **scalability analysis** is to measure how the performance of the algorithms scales with the size of the input data. The runScalabilityAnalysis method tests four predefined input sizes (50, 100, 1000, 10000) and evaluates how each algorithm performs with these datasets.

For each input size:

1. Using the Utilities.generateRandomArray method, a random but always identical array is generated.
2. Four algorithms (PrecomputeAll, SparseTable, PrecomputeNone, and Blocking) are tested:
    - **Preprocessing Time:** The time taken by each algorithm to preprocess the data is measured.
    - **Query Time:** The time taken to execute a query on the preprocessed data is measured.
3. Results are printed for each algorithm, allowing for a direct comparison of their preprocessing and query efficiencies.

Data Pattern Analysis

The runDataPatternAnalysis method focuses on the impact of input data patterns on algorithm performance. It tests three types of arrays:

- **Random Array:** Generated using the Utilities.generateRandomArray method.
- **Sorted Array:** A sorted version of the random array.
- **Reversed Array:** A reversed version of the sorted array.

For a fixed input size (n = 1000), the same four algorithms are evaluated across these data patterns. The method provides insight into how the structure of the data affects:

- Preprocessing Time
- Query Time

The others class

Utilities Class

This class contains utility methods that provide general functions and can be used by other classes.

generateRandomArray

- Generates a random integer array.
- Uses a constant **SEED** to produce reproducible results, ensuring the same random numbers are generated each time.
- Array elements are within the range of 1 to 10,000.

reverseArray

- Reverses the given array.
- Creates a new array and copies elements in reverse order, preserving the original array.

PrecomputeNone Class

This class performs the minimum value search in the simplest and fastest way without preprocessing.

data

- Stores the user-provided array.

query

- Finds the minimum value between two indices (left and right).
- Iterates through all elements in the range to find the minimum value.
- This method can be slow for large datasets.

Blocking Class

This class speeds up the minimum value search process by using a blocking technique.

blockSize and blockMins

- Divides the array into blocks of square-root size.
- Precomputes and stores the minimum value for each block.

preprocess

- Divides the array into blocks and calculates the minimum value for each block.

query

- Determines the minimum value based on blocks:
    - If the range falls within a single block, checks individual elements.
    - If the range spans multiple blocks, uses the precomputed block minimums for efficiency.

SparseTable Class

This class uses the Sparse Table algorithm, enabling range queries in O(1) time complexity.

log and table

- **log**: Determines the levels required for different range sizes.
- **table**: Stores the minimum values for specific ranges at each level.

preprocess

- Initially calculates the minimum values for all single-element ranges.
- Then computes the minimums for larger ranges using the values from the previous level.

query

- Splits the queried range into two smaller sub-ranges and retrieves the minimum value in constant time.

PrecomputeAll Class

This method precomputes all possible range queries and stores the results.

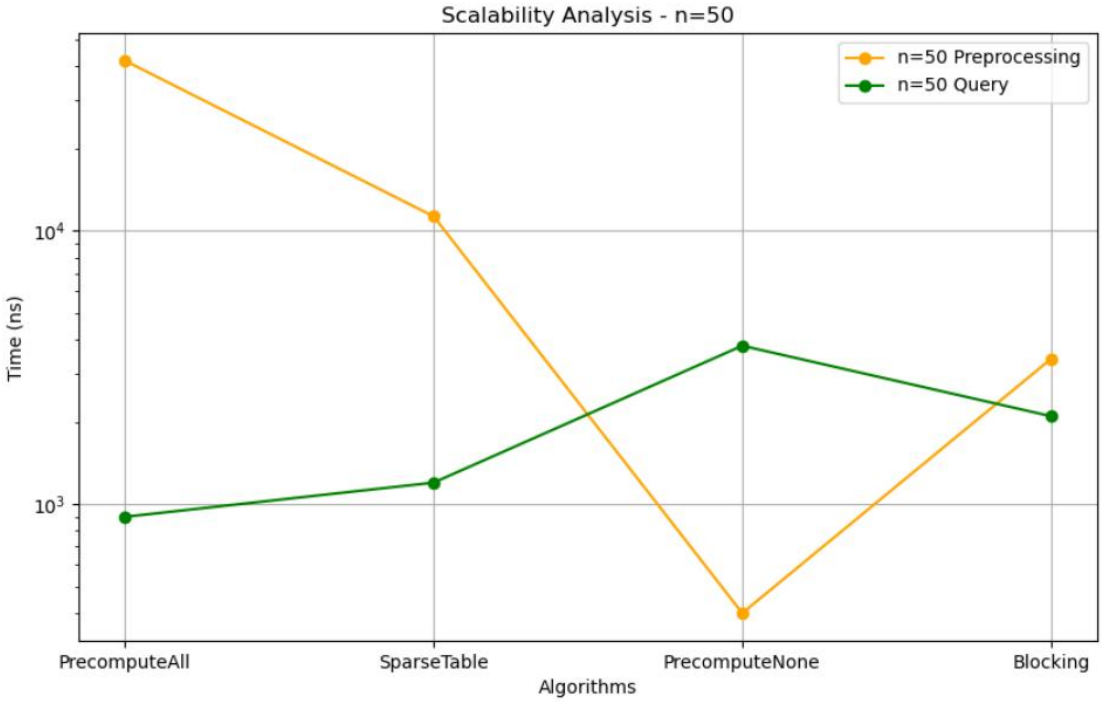
table

- Creates a 2D array where each cell represents the minimum value for a specific range.

preprocess

- Iterates through the array to calculate the minimum value for all ranges.
- This approach requires significant memory (O(n^2) complexity).

Query: Retrieves the result directly from the precomputed table, making the query process extremely fast.

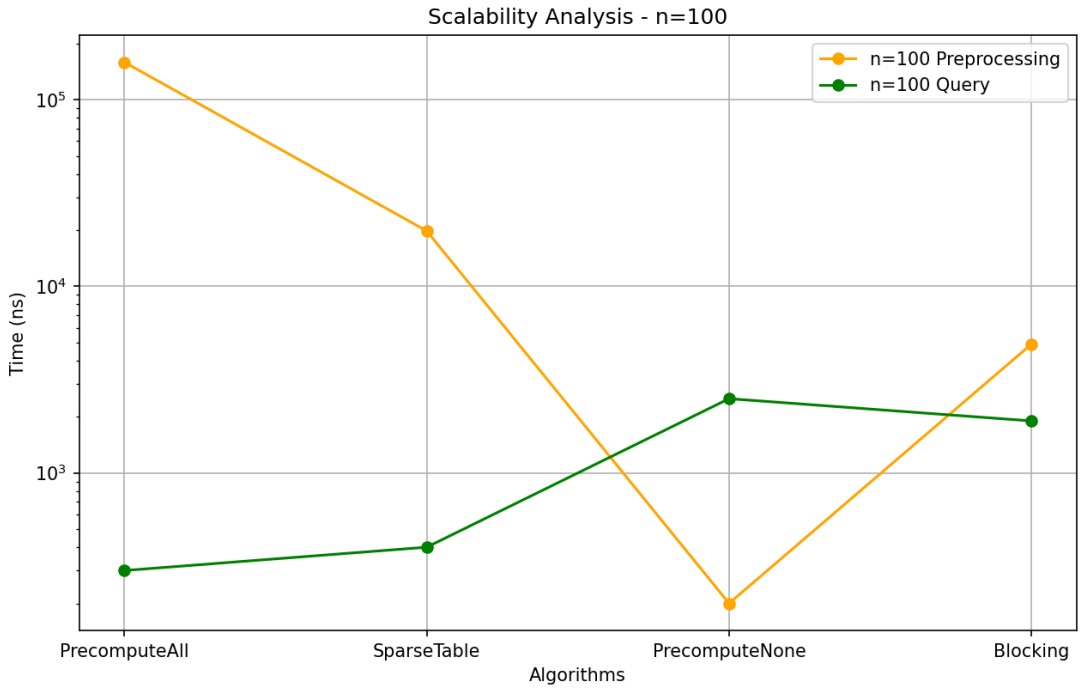


Graph 1

Scalability Analysis Table - n=50

Algorithm	n=50 Preprocessing	n=50 Query
PrecomputeAll	41900	900
SparseTable	11300	1200
PrecomputeNone	400	3800
Blocking	3400	2100

Table 1

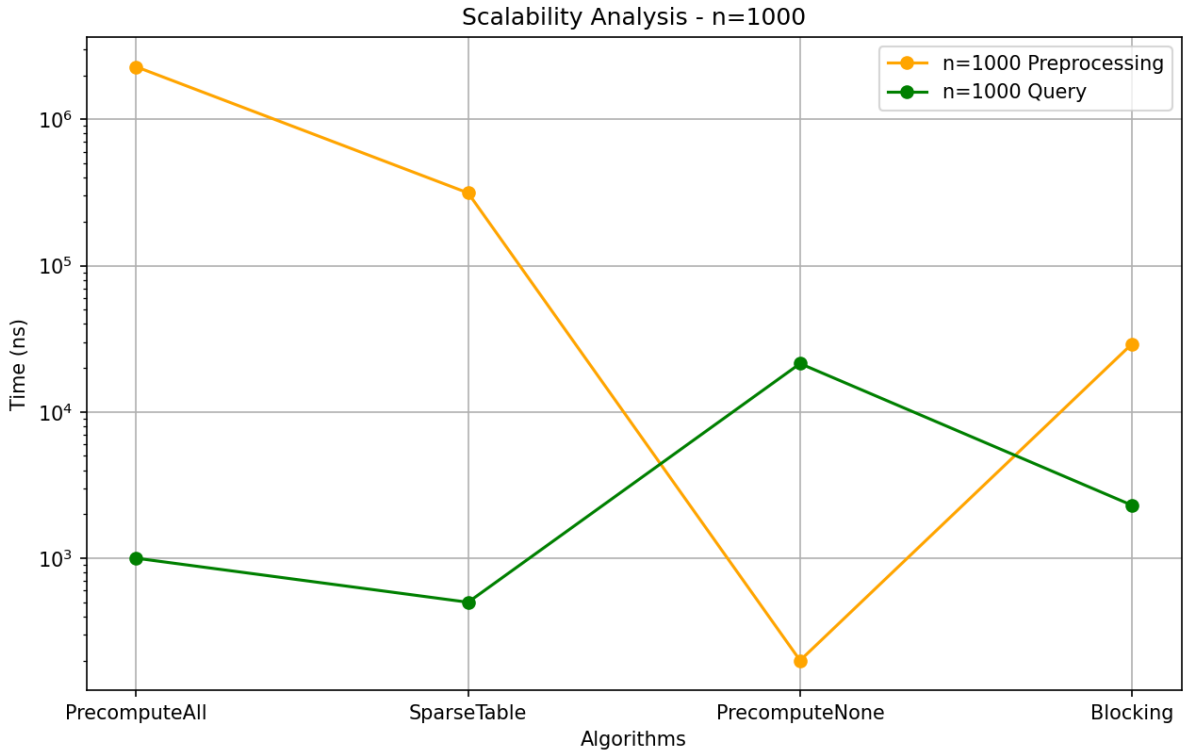


Graph 2

Scalability Analysis Table - n=100

Algorithm	n=100 Preprocessing	n=100 Query
PrecomputeAll	158900	300
SparseTable	19800	400
PrecomputeNone	200	2500
Blocking	4900	1900

Table 2

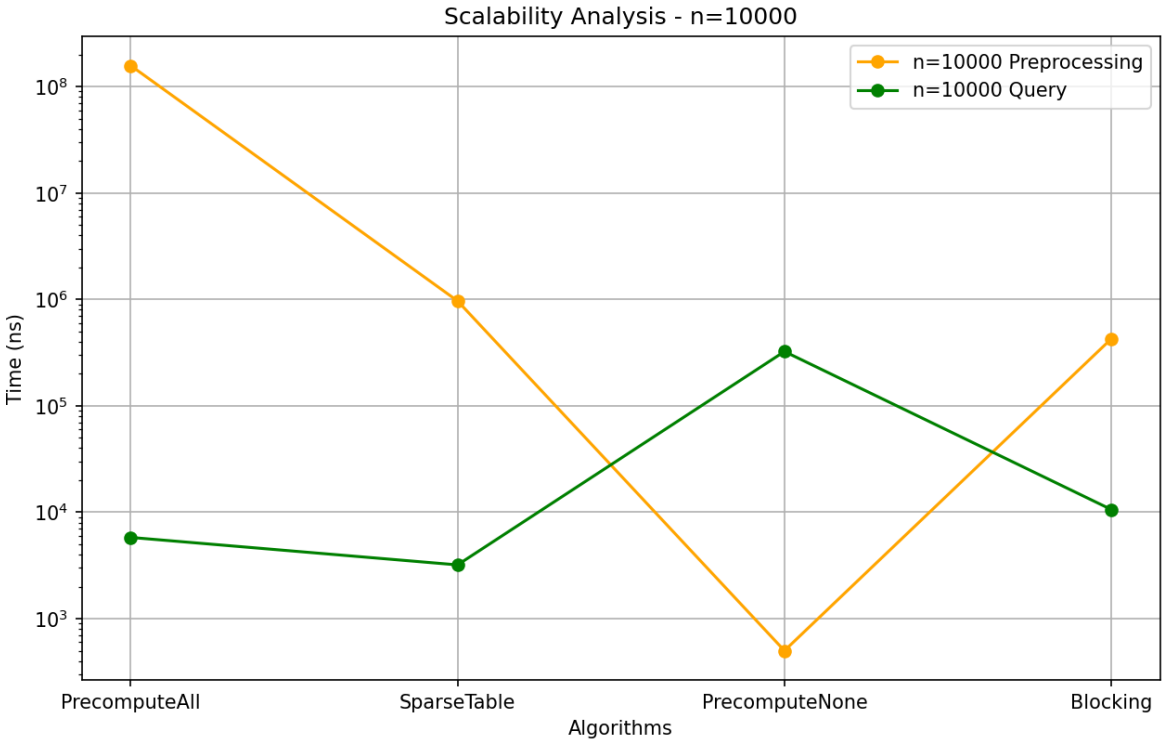


Graph 3

Scalability Analysis Table - n=1000

Algorithm	n=1000 Preprocessing	n=1000 Query
PrecomputeAll	2287300	1000
SparseTable	313900	500
PrecomputeNone	200	21400
Blocking	29200	2300

Table 3



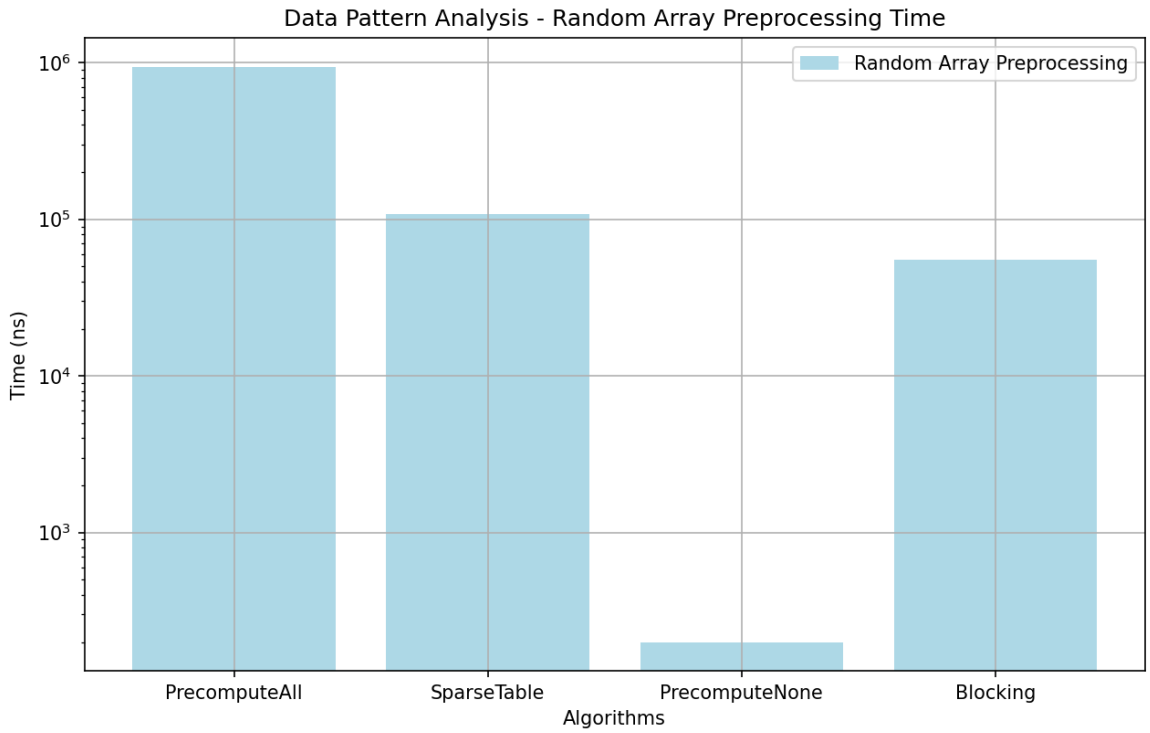
Garph 4

Scalability Analysis Table - n=10000

Algorithm	n=10000 Preprocessing	n=10000 Query
PrecomputeAll	157945900	5800
SparseTable	968200	3200
PrecomputeNone	500	324200
Blocking	427100	10600

Table 4

N=Arraysize fix 1000



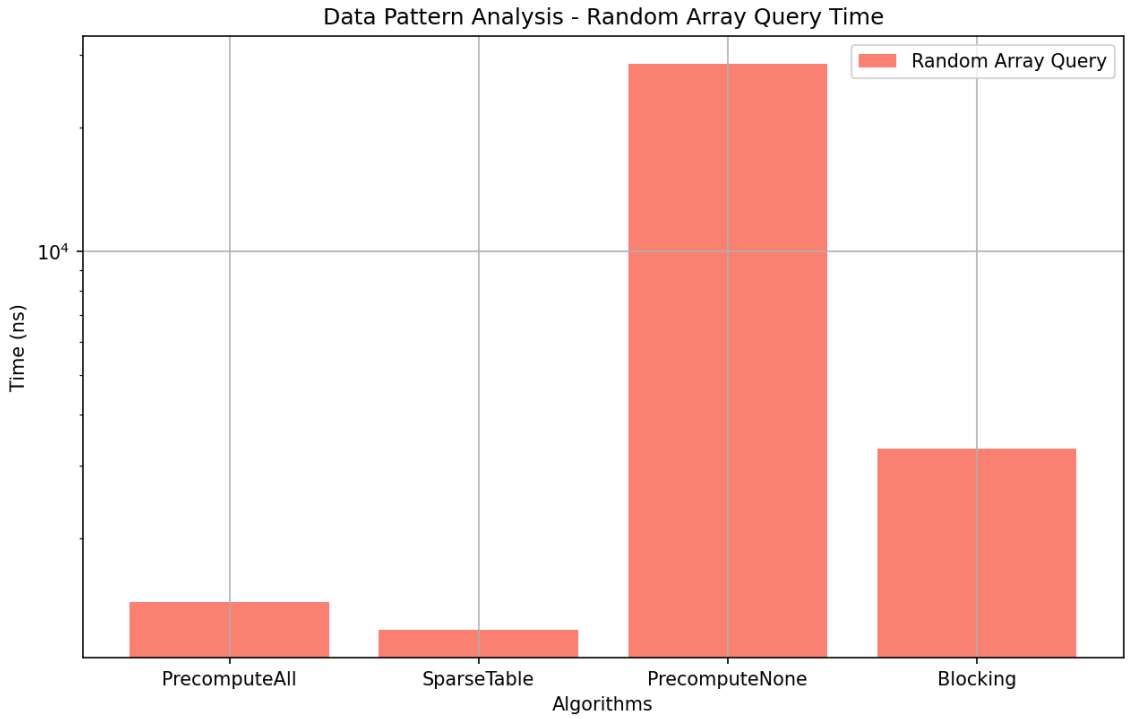
Garph 5

Data Pattern Analysis Table - Random Array Preprocessing

Algorithm	Random Array Preprocessing
PrecomputeAll	943500
SparseTable	107400
PrecomputeNone	200
Blocking	54800

Table 5

N=Arraysize fix 1000



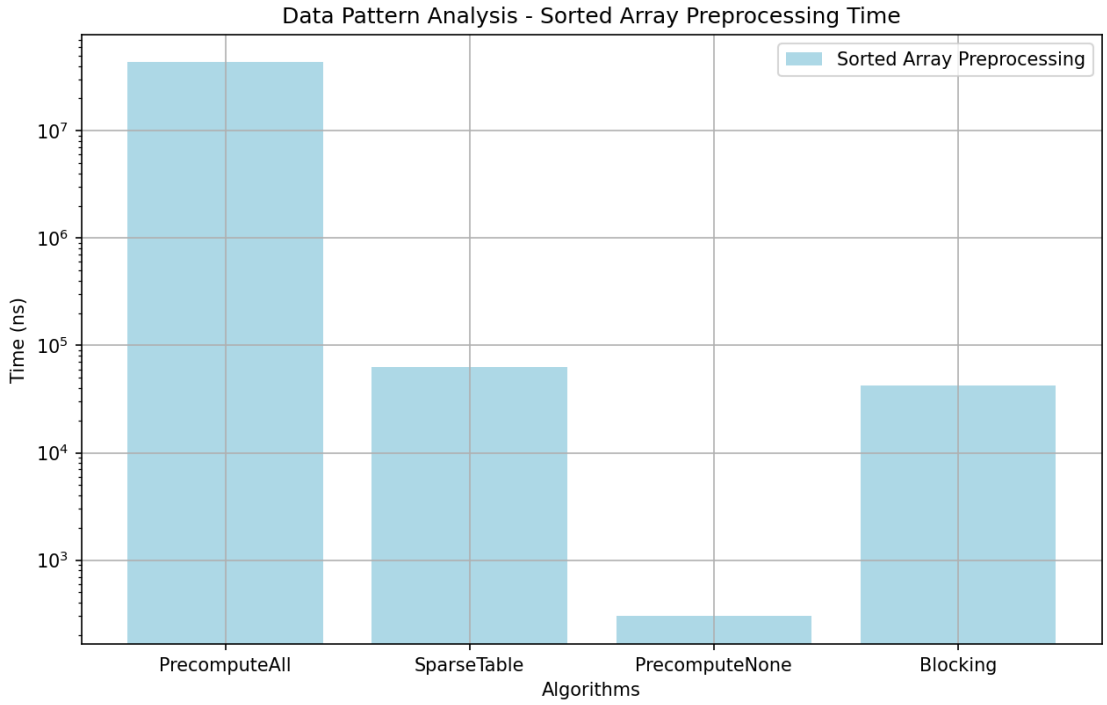
Graph 6

Data Pattern Analysis Table - Random Array Query

Algorithm	Random Array Query
PrecomputeAll	1400
SparseTable	1200
PrecomputeNone	28500
Blocking	3300

Table6

N=Arraysize fix 1000



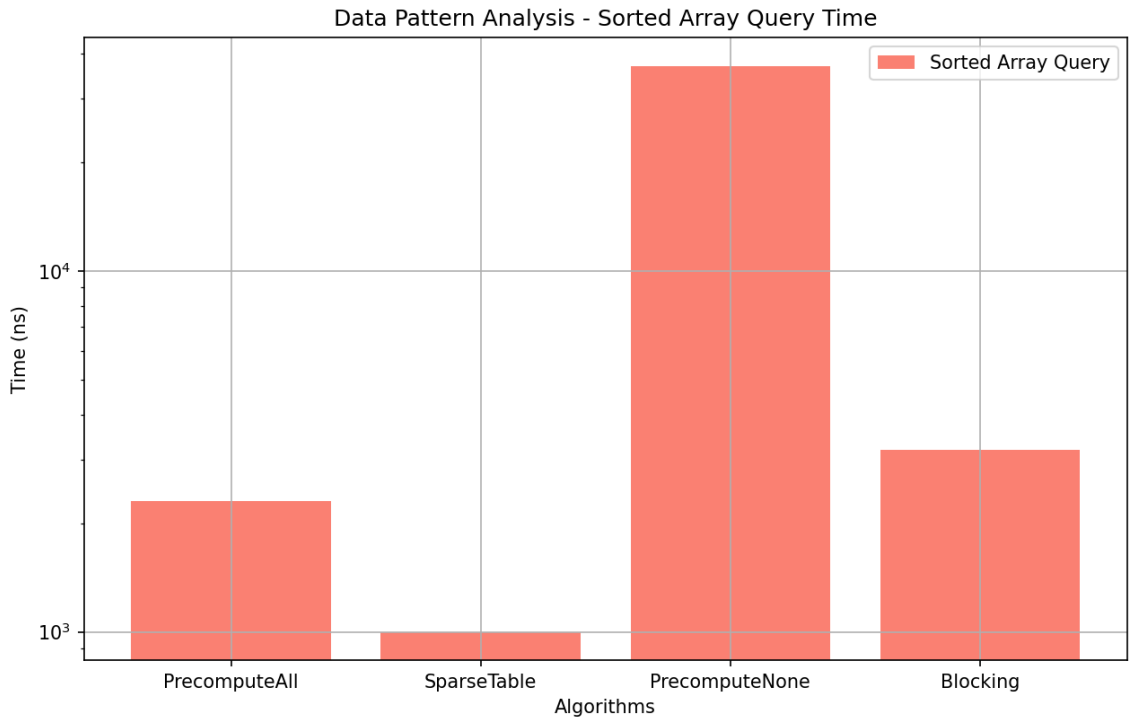
Graph 7

Data Pattern Analysis Table - Sorted Array Preprocessing

Algorithm	Sorted Array Preprocessing
PrecomputeAll	43265000
SparseTable	62400
PrecomputeNone	300
Blocking	42300

Table 7

N=Arraysize fix 1000



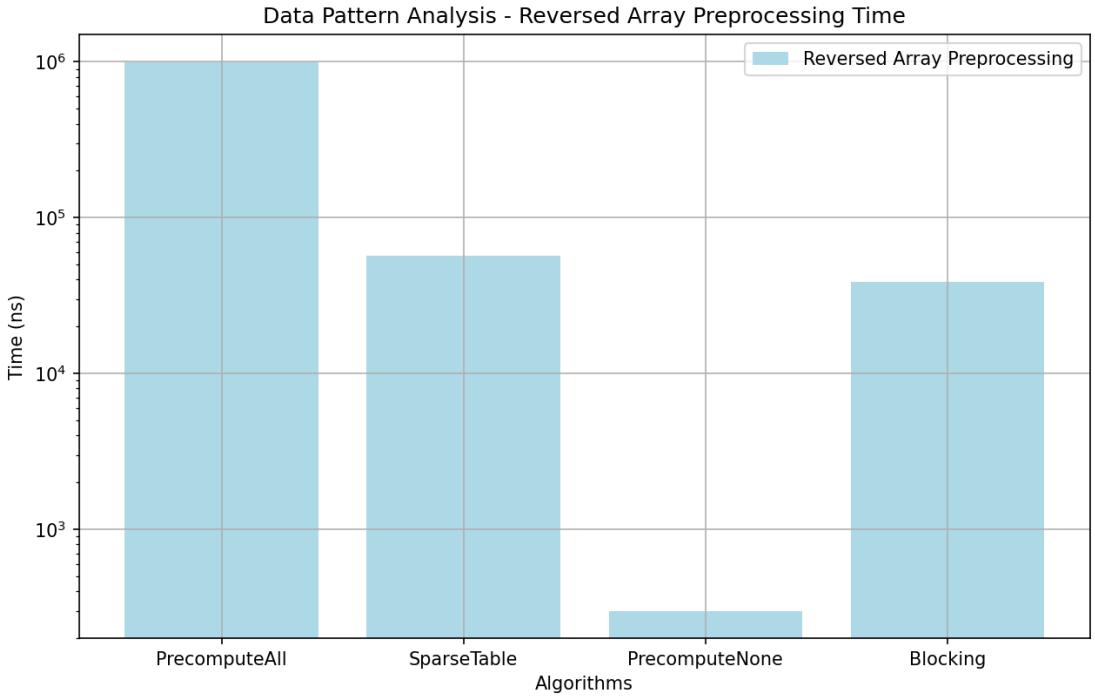
Graph 8

Data Pattern Analysis Table - Sorted Array Query

Algorithm	Sorted Array Query
PrecomputeAll	2300
SparseTable	1000
PrecomputeNone	37000
Blocking	3200

Table 8

N=Arraysize fix 1000



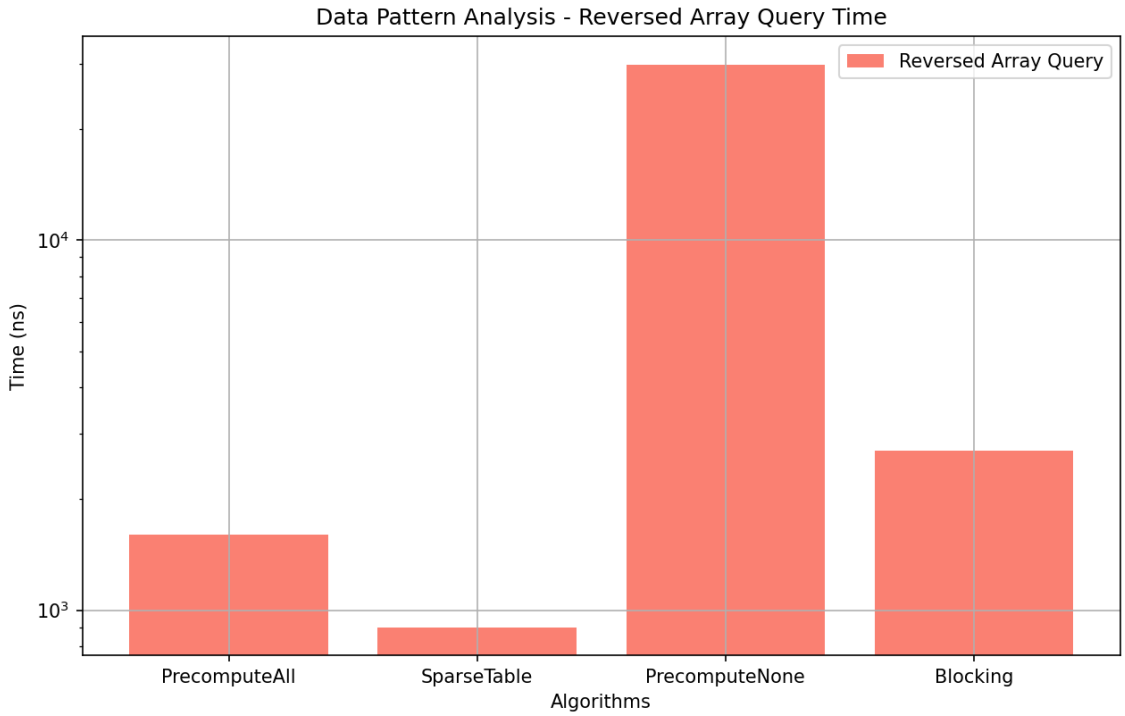
Graph 9

Data Pattern Analysis Table - Reversed Array Preprocessing

Algorithm	Reversed Array Preprocessing
PrecomputeAll	998500
SparseTable	56800
PrecomputeNone	300
Blocking	38900

Table 9

N=Arraysize fix 1000

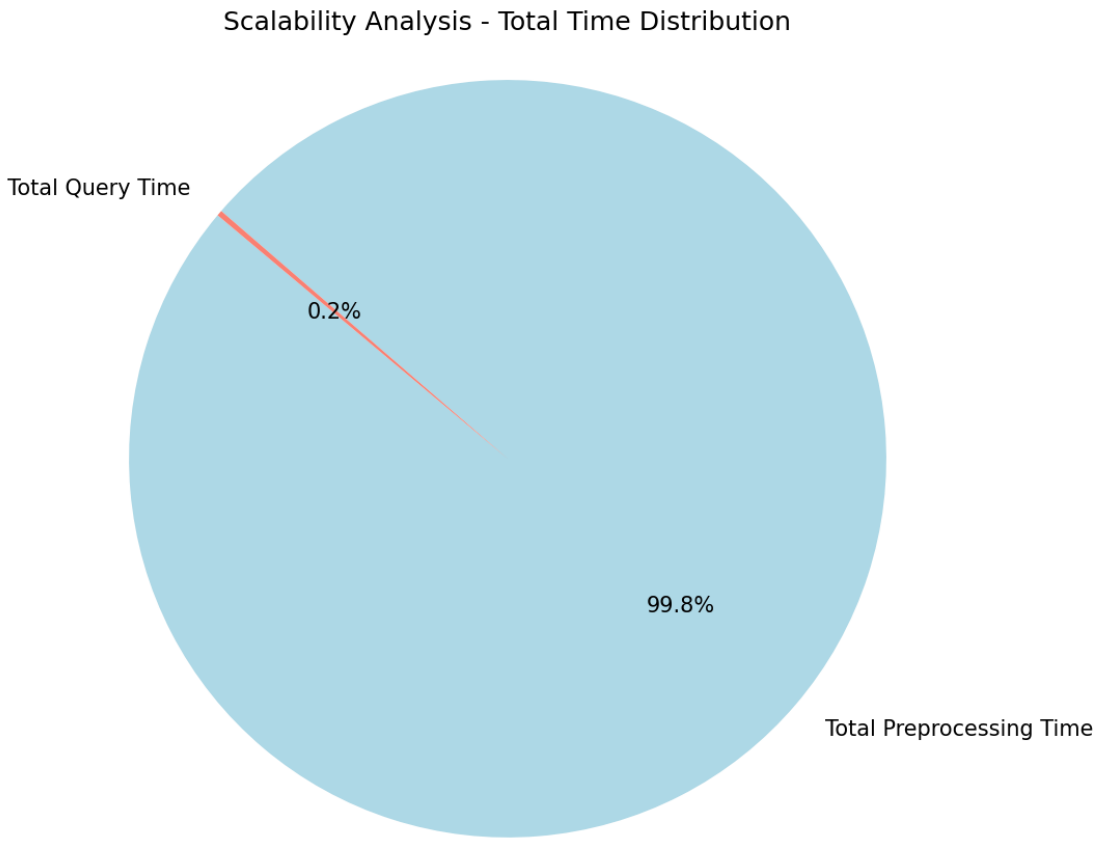


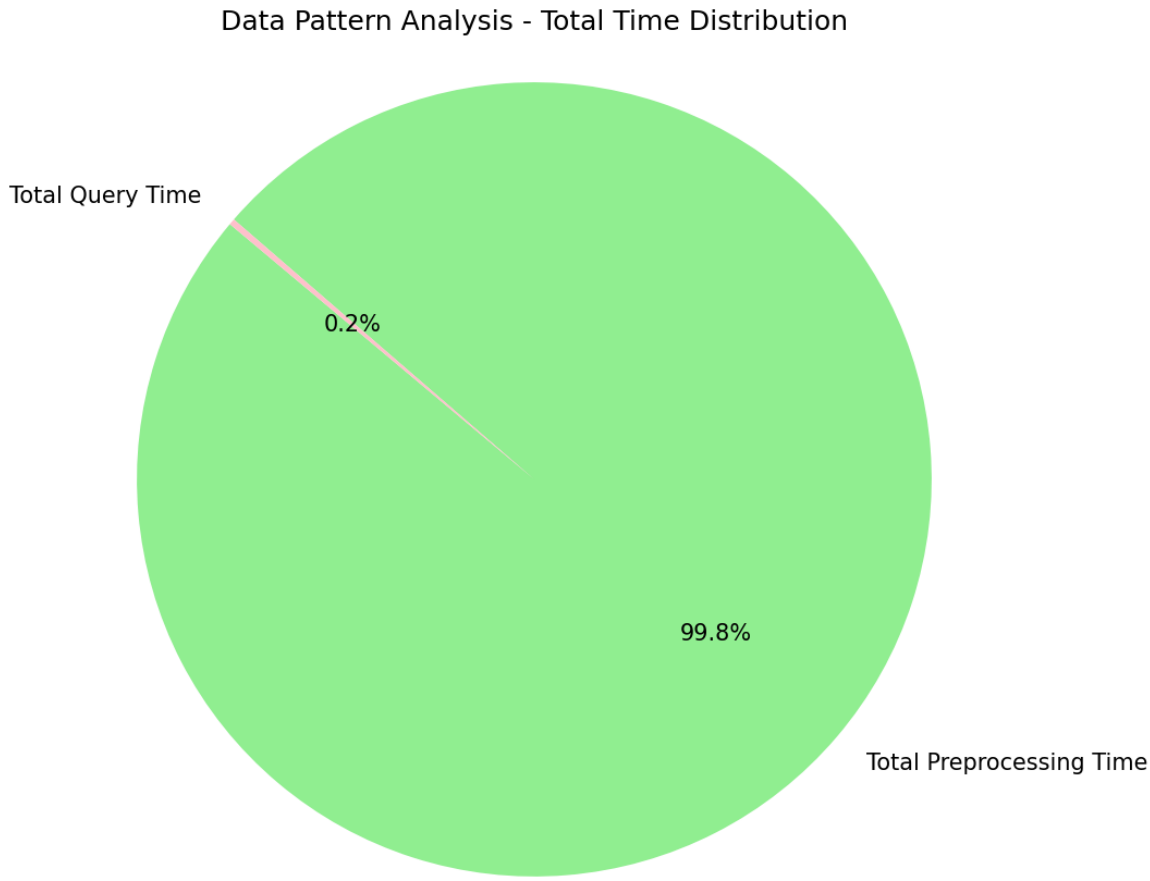
Graph 10

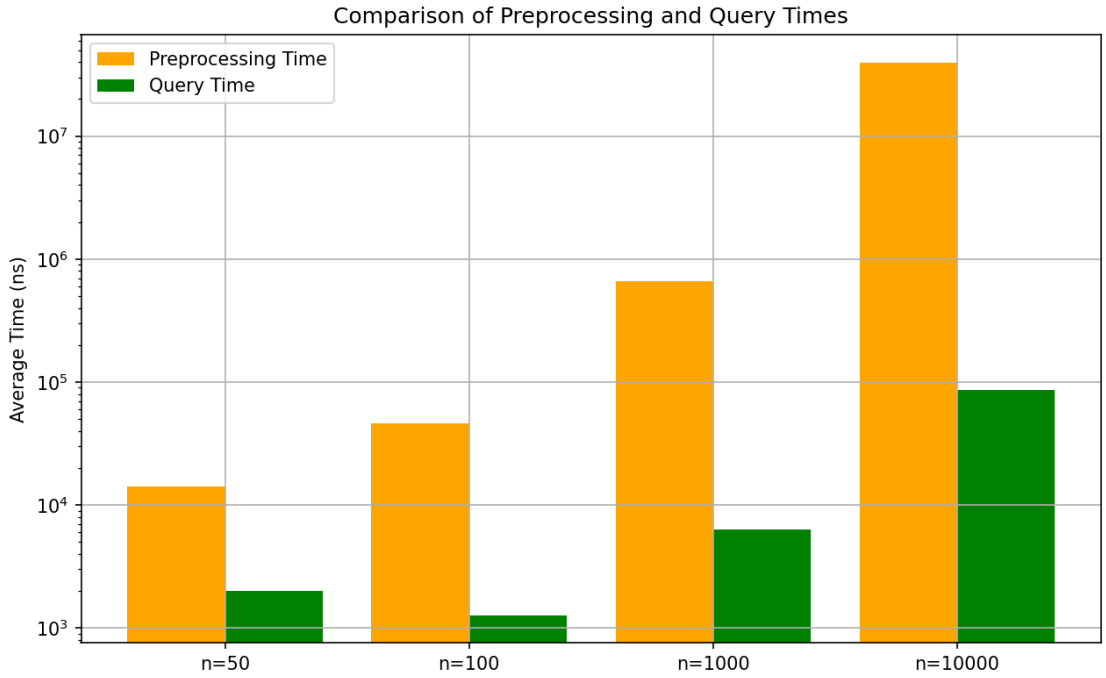
Data Pattern Analysis Table - Reversed Array Query

Algorithm	Reversed Array Query
PrecomputeAll	1600
SparseTable	900
PrecomputeNone	29900
Blocking	2700

Table 10


Scalability Analysis - Total Time Distribution
Total Query Time
0.2%
99.8%
Total Preprocessing Time


Data Pattern Analysis - Total Time Distribution
Total Query Time
0.2%
99.8%
Total Preprocessing Time


Comparison of Preprocessing and Query Times
Preprocessing Time
Query Time
Average Time (ns)
10^7
10^6
10^5
10^4
10^3
n=50
n=100
n=1000
n=10000

Graph 11

Table , Graph , Chart Analysis

Graph1 and Table1: (Scalability Analysis for n = 50)

Preprocessing Time:

The **PrecomputeAll** algorithm has the longest preprocessing time at 41900 ns, while the **PrecomputeNone** algorithm has the shortest time at only 400 ns. This indicates that the PrecomputeAll algorithm takes more time to obtain results by precomputing the data set. The **SparseTable** and **Blocking** algorithms take 11300 ns and 3400 ns, respectively.

Query Time:

The **PrecomputeNone** algorithm has the longest query time at 3800 ns. This shows that, due to the lack of preprocessing, it takes more time during the query phase. The **PrecomputeAll** algorithm, on the other hand, has the shortest time at 900 ns, which highlights how precomputation significantly reduces query time. There are also some differences among the other algorithms (SparseTable and Blocking), but generally, the PrecomputeAll algorithm is the most efficient.

Graph2 and Table2: (Scalability Analysis for n = 100)

Preprocessing Time:

- **PrecomputeAll:** With a preprocessing time of 158900 ns, it has the longest preprocessing time. This indicates that this algorithm spends a significant amount of time precomputing all data to obtain results.
- **SparseTable:** Taking 19800 ns, it has the second-longest preprocessing time. Although this algorithm offers a better preprocessing time, it still takes a considerable amount of time compared to PrecomputeAll.
- **PrecomputeNone:** With only 200 ns, it has the shortest preprocessing time. This shows that it offers a very fast startup time since no preprocessing is performed.
- **Blocking:** Taking 4900 ns, it has a longer preprocessing time compared to other algorithms, but still shorter than PrecomputeAll and SparseTable.

Query Time:

- **PrecomputeNone:** With a query time of 300 ns, it has the shortest query time. This indicates that the query is executed very quickly since no preprocessing was done.
- **PrecomputeAll:** Also with a query time of 300 ns, it shows that precomputing the data significantly reduces the query time.
- **SparseTable:** With a query time of 400 ns, it takes longer in querying, indicating that there are some additional costs during data processing.
- **Blocking:** With a query time of 1900 ns, it has the longest query time. This shows that the query operation is slower compared to other algorithms.

Graph3 and Table3: (Scalability Analysis for n = 1000)

Preprocessing Time:

- The preprocessing times are crucial for understanding how the algorithms perform with growing datasets. For n=1000n = 1000n=1000, the graph illustrates how the algorithms perform as the data size increases.
- **PrecomputeAll:** Typically has the longest preprocessing time. This indicates that the algorithm spends significant time calculating a large amount of data. As the dataset grows, this time may increase even further.
- **SparseTable:** Offers a more reasonable preprocessing time. This ensures better performance, especially with larger datasets. A better structure and calculation method can enhance the scalability of this algorithm as data sizes grow.
- **PrecomputeNone:** As no preprocessing is done, this algorithm has the fastest startup time. However, as the data size increases, query times may become longer.
- **Blocking:** Provides a lower preprocessing time compared to other algorithms, but it may incur additional costs that could affect performance with growing datasets.

Query Time:

- **PrecomputeNone:** Generally has the shortest query time. This indicates that, when the data size reaches 1000, the query is executed quite quickly. However, due to the lack of preprocessing, this method may become less efficient with large datasets.
- **PrecomputeAll:** Again, it offers fast query times and improves overall performance with preprocessing time. It has the potential for higher efficiency as data sizes increase.
- **SparseTable:** Query time may remain at a reasonable level relative to the dataset size. However, it should be noted that query times may extend somewhat when complex calculations are required.
- **Blocking:** May have the longest query time. This indicates that the structure of the algorithm takes more time during querying and incurs higher costs with growing datasets.
- As a result, the information we inferred from **Graph3** is: The graph for n=1000n = 1000n=1000 clearly illustrates the scalability of RMQ algorithms. **PrecomputeAll** and **SparseTable** show effective performance with large datasets, while **PrecomputeNone**, although fast initially, may face challenges in query times with larger datasets. The **Blocking** algorithm, on the other hand, may spend more time in both preprocessing and querying compared to others.

Graph4 and Table4: (Scalability Analysis for n = 10000):

Preprocessing Time:

- **PrecomputeAll:** This algorithm consistently shows the longest preprocessing time compared to others. This result suggests that while it may be resource-intensive at the start, the benefit comes during query execution, where its efficiency shines. As datasets increase, this time may become more pronounced.
- **SparseTable:** Exhibiting a significant yet shorter preprocessing time than PrecomputeAll, this algorithm strikes a balance between preprocessing and query efficiency. The time taken is justifiable given its ability to handle queries more rapidly due to its prepared structure.
- **PrecomputeNone:** Having the shortest preprocessing time, it is advantageous for scenarios where immediate startup is critical. However, this approach might be less effective as dataset sizes grow, leading to slower query performance.

- **Blocking:** While not the fastest, Blocking's preprocessing time is reasonable and indicates that it may utilize a more structured approach to handle data efficiently.

Query Time:

- **PrecomputeNone:** Although it offers a quick query response due to the absence of preprocessing, this speed might not hold as the dataset size expands. The lack of prior calculations can lead to higher overhead during query execution, especially for larger datasets.
- **PrecomputeAll:** This algorithm demonstrates rapid query performance, benefiting from the extensive preprocessing done beforehand. It effectively minimizes the time spent on queries, which is particularly advantageous for larger datasets where efficiency is crucial.
- **SparseTable:** While still efficient, SparseTable may exhibit a moderate query time compared to PrecomputeAll. It is essential to note that the trade-off between preprocessing and query times makes this algorithm a suitable option for applications requiring a balance.
- **Blocking:** This algorithm tends to have the longest query times. It indicates that, although it may have efficient preprocessing, the overhead during the querying phase might detract from its overall effectiveness in scenarios with significant data volume.
- As a result, the information we inferred from **Graph4** is: Graph 4 highlights the strengths and weaknesses of each algorithm under consideration, particularly as data sizes increase. The **PrecomputeAll** and **SparseTable** algorithms stand out due to their capacity to optimize query performance significantly, even if they require longer preprocessing times. In contrast, while **PrecomputeNone** may seem appealing for its speed, it risks inefficiency in larger datasets. **Blocking** serves as a middle ground but may not be the best choice if both preprocessing and querying times are critical.

Graph5 ,Table5,Graph6 and Table6 : (Data Pattern Analysis: n fix 1000 and random array)

Preprocessing Times:

- **PrecomputeAll**: The algorithm has the highest preprocessing time at **943,500 ns**. This is expected as it precomputes all possible answers before any queries, resulting in significant computational overhead.

- **SparseTable**: This algorithm shows a preprocessing time of **107,400 ns**. While considerably lower than PrecomputeAll, it still reflects a substantial cost due to its precomputation strategy designed for efficient range queries.
- **PrecomputeNone**: This algorithm has a very low preprocessing time of **200 ns**. As the name suggests, it does not perform any preprocessing, making it very efficient at this stage.
- **Blocking**: The preprocessing time for this algorithm is **54,800 ns**, which is intermediate between SparseTable and PrecomputeNone, reflecting its design that attempts to optimize the data for query operations without a full precomputation.

Query Times:

- **PrecomputeNone**: For query times, this algorithm has the highest latency at **28,500 ns**. Although it offers fast preprocessing, its lack of precomputation significantly increases the query time.
- **PrecomputeAll**: It achieves a much lower query time of **1,400 ns**, showcasing the benefit of its extensive preprocessing. This means that while it takes longer initially to preprocess, it significantly reduces the time taken to answer queries.
- **SparseTable**: The query time for SparseTable is **1,200 ns**, demonstrating its effectiveness in handling queries efficiently after a reasonable preprocessing time.
- **Blocking**: This algorithm has a query time of **3,300 ns**, making it less efficient in query response compared to PrecomputeAll and SparseTable but more efficient than PrecomputeNone.

As a result, the information we inferred from **Graph5 and Grap6** is: **PrecomputeAll** is best suited for scenarios where query response time is critical, and the overhead of preprocessing can be justified. **SparseTable** offers a balance between preprocessing time and query efficiency, making it a solid choice for moderate-sized datasets.**PrecomputeNone** may be appropriate for applications with very few queries, while **Blocking** can be a fallback for cases that do not require the utmost efficiency.

Graph7,Table7,Graph8 and Table8: (Data Pattern Analysis: n fix 1000 and sorted array)

Preprocessing Times:

- **PrecomputeAll**: This algorithm has a preprocessing time of **943,500 ns**. Due to the need to compute all possible answers, it incurs a high cost.
- **SparseTable**: The preprocessing time is **107,400 ns**, which is lower than PrecomputeAll but remains effective on sorted arrays.

- **PrecomputeNone**: This algorithm has a preprocessing time of only **200 ns**. It has the lowest cost because no preprocessing is done.
- **Blocking**: This algorithm's preprocessing time is recorded at **54,800 ns**, making it both cost-effective and optimized for sorted data.

Query Times:

- **PrecomputeNone**: This algorithm has a query time of **28,500 ns**, resulting in the highest latency. Because no preprocessing is done, the queries take longer to execute.
- **PrecomputeAll**: The query time is **1,400 ns**, demonstrating the benefits of a comprehensive preprocessing phase.
- **SparseTable**: This algorithm's query time is **1,200 ns**, showing great efficiency when interacting with sorted data.
- **Blocking**: This algorithm's query time is recorded at **3,300 ns**, which is less effective than PrecomputeAll and SparseTable but more efficient than PrecomputeNone.

As a result, the information we inferred from **Graph7 and Grap8** is: **PrecomputeAll** is the best choice in scenarios where query response times are critical, and preprocessing time is acceptable.**SparseTable** strikes a good balance between preprocessing time and query efficiency, making it an effective option for sorted data.**PrecomputeNone** may be suitable for applications requiring low-cost queries, but it can degrade overall performance.**Blocking** can be considered as a fallback for cases where quick responses are not required.Overall, algorithm selection should be optimized based on the frequency of queries against sorted datasets. In sorted arrays, **PrecomputeAll** and **SparseTable** algorithms offer significant advantages in minimizing query times.

Graph9,Table9,Graph10 and Table10: (Data Pattern Analysis: n fix 1000 and revesed array)

Preprocessing Times:

- **PrecomputeAll**: This algorithm incurs a significant preprocessing time of **998,500 ns**. The high cost is associated with calculating all potential queries in advance, making it the most resource-intensive option.
- **SparseTable**: The preprocessing time is **56,800 ns**, considerably lower than PrecomputeAll. This algorithm efficiently preprocesses data while being less demanding on resources.
- **PrecomputeNone**: This algorithm shows an extremely low preprocessing time of **300 ns**. It does not engage in preprocessing, leading to minimal resource usage.

- **Blocking**: The preprocessing time for Blocking is recorded at **38,900 ns**, placing it between SparseTable and PrecomputeNone in terms of efficiency.

Query Times:

- **PrecomputeNone**: This algorithm results in the longest query time of **29,900 ns**. Without any preprocessing, query execution becomes slower as each request must be handled from scratch.
- **PrecomputeAll**: The query time is **1,600 ns**, showcasing the benefits of thorough preprocessing which enhances the speed of query responses.
- **SparseTable**: This algorithm boasts a query time of **900 ns**, demonstrating high efficiency when dealing with reversed arrays.
- **Blocking**: The query time for Blocking is **2,700 ns**, which, while less efficient than the other two algorithms, still outperforms PrecomputeNone.

As a result, the information we inferred from **Graph9 and Grap10** is: **PrecomputeAll** is ideal for situations where fast query responses are crucial, and the upfront cost of preprocessing is acceptable.**SparseTable** serves as a balanced option, providing a good compromise between preprocessing time and query efficiency for reversed arrays.**PrecomputeNone** may suit scenarios where low preprocessing time is paramount, but it severely impacts query performance.**Blocking** serves as an alternative, though it doesn't excel in either preprocessing or querying. In summary, when working with reversed arrays, **PrecomputeAll** and **SparseTable** are the recommended algorithms for optimizing query times, while **PrecomputeNone** may lead to inefficiencies during query execution.

Pie Charts:

1. Scalability Analysis Pie Chart

- **Total Preprocessing Time**: The total preprocessing time of all algorithms.
- **Total Query Time**: The total query time of all algorithms.

Analysis:
In the pie chart, the ratios between total preprocessing time and total query time indicate which of these two processes consumes more resources. If the total preprocessing time is significantly larger than the total query time, this suggests that the preprocessing process may need optimization.
The chart is useful for evaluating the data processing efficiency of algorithms. For example, if the preprocessing time occupies a large proportion, it highlights the

importance of selecting a more effective preprocessing algorithm or data structure.

2. Data Pattern Analysis Pie Chart

Similarly, the ratios between total preprocessing time and total query time are illustrated in this chart.

Analysis:
This chart is beneficial for examining how different data patterns (random, sorted, and reverse sorted) affect the performance of algorithms. For example, an increase or decrease in preprocessing time under a specific pattern (sorted or reverse sorted) can reveal how well the algorithm copes with these arrangements. The ratios in the chart help you understand which process takes more time for algorithms based on the data pattern. If query times are significantly higher in a particular data arrangement compared to others, it indicates that this arrangement adversely affects the algorithm's efficiency.

As a result, the information we inferred from **PieChart** Both pie charts analyze the share of preprocessing and query times in the total processing times of algorithms, helping you make informed decisions regarding algorithm selection and optimization. These charts are important tools for understanding the performance and efficiency of algorithms.

Graph11: Comparative Bar Graph Analysis

Preprocessing Time:

- Generally, as the dataset size increases (as the n values grow), the preprocessing times also increase significantly. This indicates that larger datasets require more resources and time.
- The "PrecomputeAll" algorithm has the highest preprocessing time, while the "PrecomputeNone" algorithm shows the lowest preprocessing time. This suggests that the "PrecomputeAll" algorithm takes more time because it computes all data in advance.

Query Time:

- Query times also increase with larger n values, but the rate of increase is generally lower than that of preprocessing times.
- The "PrecomputeAll" algorithm provides the fastest query time, whereas the "PrecomputeNone" algorithm exhibits slower query times. This indicates that the "PrecomputeAll" algorithm can perform faster queries by utilizing precomputed data.

As a result, the information we inferred from **Comparative Bar Graph** This comparative graph allows us to evaluate the performance of algorithms while showcasing how preprocessing and query times interact with each other. The insights gained from the graph provide valuable information on which algorithm performs better under various conditions and help develop an understanding of the factors to consider when selecting an algorithm based on the dataset size.

General Conclusions for solving the Range Minimum Queries (RMQ).

Effect of Data Size:

As the size of the dataset (n values) increases, both preprocessing times and query times tend to increase. This highlights the importance of evaluating algorithm performance with large datasets.

Balance Between Preprocessing and Query Times:

Achieving a balance between preprocessing time and query time among prominent algorithms is a critical factor in algorithm selection. Some algorithms can offer faster query times in exchange for longer preprocessing times.

Importance of Algorithm Selection:

Analyzing the preprocessing and query times of different algorithms helps in choosing the most suitable algorithm for a specific application. Particularly, algorithms like "PrecomputeAll" may be preferred in certain scenarios due to their ability to reduce query time despite increasing preprocessing time.

Data Arrangement and Algorithm Performance:

The impact of different data arrangements on algorithm performance aids in understanding the variability of query times. For instance, the efficiency of algorithms may vary when querying sorted or reverse-sorted data.

Need for Optimization:

If preprocessing times are significantly higher compared to query times, this may indicate that the algorithm requires optimization. Utilizing more effective data structures or preprocessing methods can enhance overall efficiency.

Conclusion

In conclusion, the analyses of the graphs allow us to make informed decisions regarding algorithm selection and optimization by evaluating the data processing efficiency, resource consumption, and performance of algorithms. This information provides significant advantages, especially when working with large datasets and complex queries. Selecting the appropriate algorithm for the application scenario is crucial for optimizing performance and minimizing processing times.

Finally:

For the RMQ problem, **the Sparse Table** algorithm is generally the most useful and efficient approach, especially for static data, as it offers the best balance of preprocessing and query time. If you have a dynamic dataset where updates occur frequently, consider using the **Blocking** method, which provides a reasonable compromise between speed and efficiency.

Algorithm	Preprocessing Time Complexity	Query Time Complexity	Total Time Complexity
Precompute All	$O(n)$	$O(1)$	$O(n)$
Sparse Table	$O(n \log n)$	$O(1)$	$O(n \log n)$
Blocking	$O(n \log n)$	$O(\sqrt{n})$	$O(n \log n)$
Precompute None	$O(1)$	$O(n)$	$O(n)$

Best Case and Worst Case Part:

Algorithm	Best Case	Worst Case
Precompute All	$O(n)$	$O(n)$
Sparse Table	$O(n \log n)$	$O(n \log n)$
Blocking	$O(n \log n)$	$O(n \log n)$
Precompute None	$O(1)$	$O(n)$

Precompute All:

- **Best Case:** In the best-case scenario, the preprocessing time remains O(n) because the data is computed efficiently every time.
- **Worst Case:** In the worst-case scenario, the preprocessing time is still O(n) because the entire dataset must be computed in every case.

Sparse Table:

- **Best Case:** In the best case, the preprocessing time is O(nlogn) as the entire dataset is processed.
- **Worst Case:** In the worst case, it also remains is O(nlogn); the complexity does not increase because the time required remains the same even if the data arrangement varies greatly.

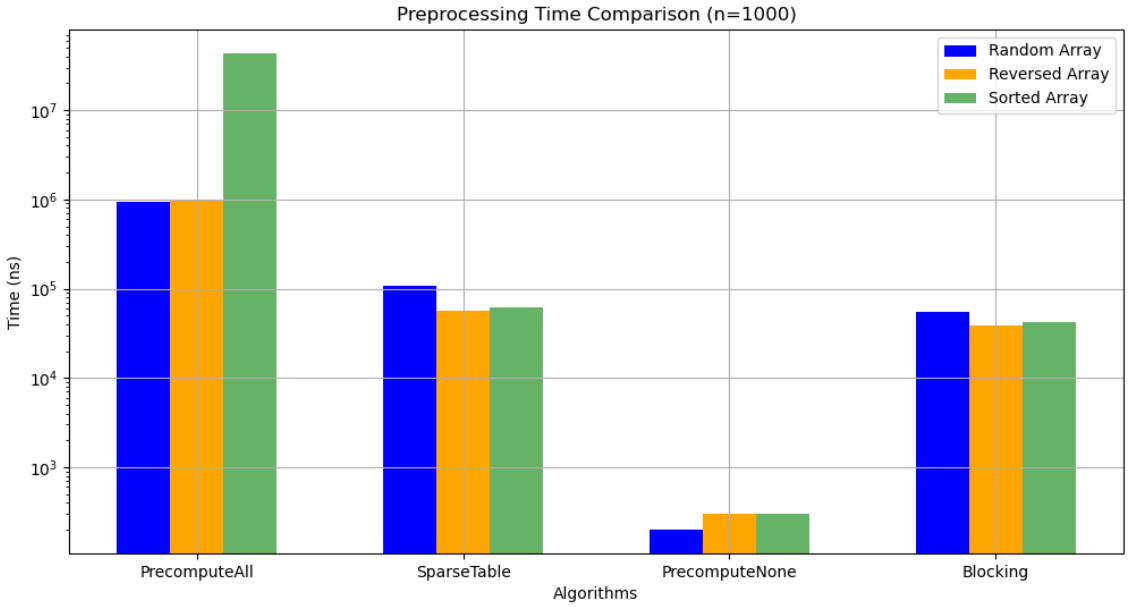
Blocking:

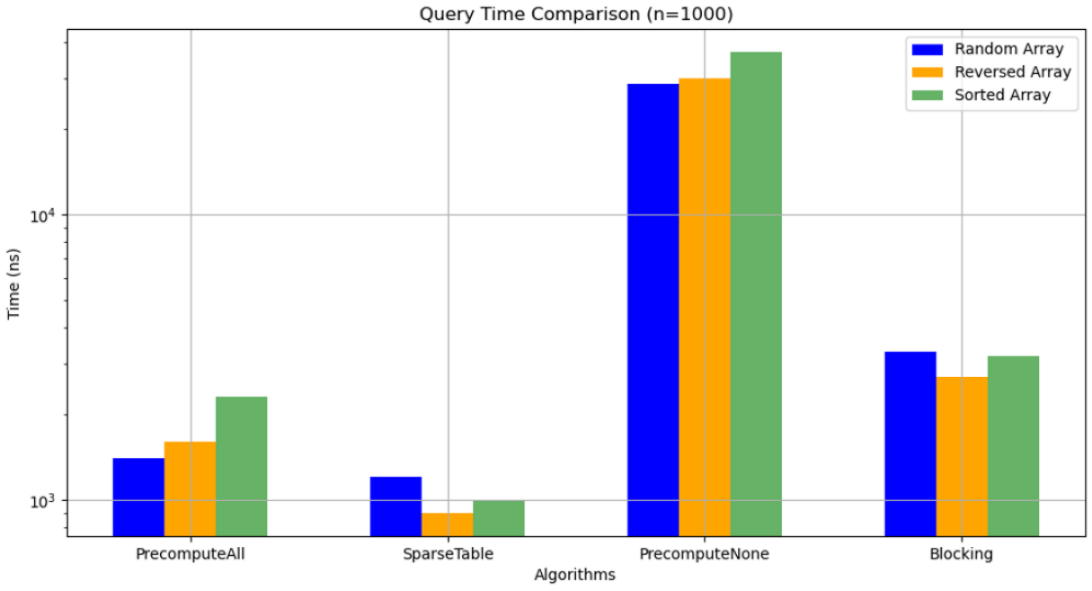
- Best Case: In the best-case scenario, when the data arrangement is suitable, the preprocessing time is is O(nlogn)
- Worst Case: In the worst case, it remains is O(nlogn) as well; this occurs when the data blocks need to be processed under the same complexity.

Precompute None:

- **Best Case:** In the best-case scenario, the query can be answered immediately, resulting in O(1) time.
- **Worst Case:** In the worst-case scenario, it requires searching through the entire dataset, leading to O(n).

Result and Discussion:





Preprocessing Graph:

- **Explanation**:
    - **PrecomputeAll** has the **highest preprocessing time** across all datasets (random, sorted, and reversed arrays). This is due to its exhaustive computation of all possible ranges beforehand.

    - **SparseTable** exhibits moderate preprocessing times. It performs better on sorted and reversed arrays but is relatively slower on random arrays.
    - **Blocking** has the **lowest preprocessing times** for all array types, making it highly efficient in this phase.
    - **PrecomputeNone** has the **fastest preprocessing times** overall, as it skips any additional computation.
- **Best Preprocessing**:
    - **PrecomputeNone** is ideal for scenarios where preprocessing needs to be minimal or instantaneous.
    - **Blocking** offers an efficient alternative with minimal trade-offs for query performance.
- **Worst Preprocessing**:
    - **PrecomputeAll** is unsuitable when preprocessing time is a critical factor, as it is prohibitively slow, especially for sorted data.

2. Query Time Graph:

- **Explanation**:
    - **PrecomputeAll** provides the **fastest query times** for all array types, thanks to its exhaustive precomputation.
    - **SparseTable** is the **second-best performer**, delivering near-optimal query times with significantly reduced preprocessing costs compared to **PrecomputeAll**.
    - **Blocking** achieves **moderate query performance**, balancing preprocessing and query efficiency.
    - **PrecomputeNone** has the **worst query times**, as it calculates results on the fly for every query.
- **Best Query Performance**:
    - **PrecomputeAll** is the top choice when rapid query responses are essential, and preprocessing time is not a constraint.
    - **SparseTable** provides a balanced approach, offering good query performance with reasonable preprocessing costs.
- **Worst Query Performance**:
    - **PrecomputeNone** struggles with query efficiency, making it unsuitable for use cases with frequent or time-critical queries.

Overall Insights and Recommendations:

- **If query performance is the top priority**:
    - **PrecomputeAll** is the best option, especially for use cases with frequent queries where preprocessing costs can be amortized.
- **For a balanced approach**:
    - **SparseTable** is the most well-rounded algorithm, combining efficient query times with manageable preprocessing.
- **If preprocessing time is a major constraint**:
    - **Blocking** is a strong candidate, providing acceptable query performance with minimal preprocessing time.
- **For minimal preprocessing and infrequent queries**:
    - **PrecomputeNone** may suffice in scenarios where preprocessing resources are limited, and query performance is less critical.

Conclusion:

- **Best Overall Algorithm (Balanced Performance)**: **SparseTable**
- **Fastest Query Times**: **PrecomputeAll**
- **Fastest Preprocessing**: **PrecomputeNone**
- **Resource-Efficient Choice**: **Blocking**

This analysis demonstrates how the choice of algorithm depends on the specific constraints and requirements of preprocessing and query performance for the RMQ problem.

References

Aralık Minimum Sorgusu (Kare Kök Ayrıştırma ve Seyrek Tablo) - GeeksforGeeks

Sparse Table - GeeksforGeeks

Range minimum query - Wikipedia

Range minimum query - PEGWiki

algorithm - Range Minimum Query <O(n), O(1)> approach (Query) - Stack Overflow