FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

Student:

Name: Beyza Dudu

Surname: KESKİN

ID Number: 2221221003

Department: Computer Engineering

Project:

Topic: Study and implement algorithms for solving the Range Minimum Queries (RMQ)

Course:

Name: BLM22311E

Instructor: Assoc. Prof. Dr. Berna KİRAZ

İçindekiler

1. Problem Description

Given a sequence of integers, usually stored in an array *A*, a range minimum query (RMQ) is a pair of indices. We assume that. The solution to the query consists in finding the minimum value that occurs in *A*, between the indices *i* and *j*.

2. Algorithms (Theoretical Section)

2.1. Precompute None

Finding the minimum in a given range can be naively performed in time linear to the query range size by traversing all the values in the range of interest.

In Range Minimum Query (RMQ), the goal is to efficiently find the minimum value in a given range [L, R] of an array. The precompute none method is the naive approach where no preprocessing is done before answering queries. Instead, each query is solved independently using a brute-force search.

2.1.1. Pseudocode

void precomputeNone (int [] arr, int L, int R)

```
int min ← int MAX
for i ← L to i ← R do
    if arr[i] < min
        min ← arr[i]
```

2.1.2. Time and Space Complexity

Preprocessing is not required for the precompute none algorithm, therefore we can assume that preprocessing time complexity is in linear time O(1).

The minimum value is found by iterating through all the elements in the specified range and comparing each one. Since we only compare the values, for n is array size, arr[0] <= l <= r <= arr[n-1], the time complexity is:

$$T = \sum_{i=l}^{r-1} 1 = (r - l + 1) = O(n)$$

Best case time complexity: O(1) if L == R (single range)
Worst case time complexity: O(n)

This algorithm also does not use any additional space (in place algorithm). Everything is done in the given array. With that, space complexity is O(1).

2.2. Precompute All

The idea of the precompute all method is precomputing all the minimum element for all possible ranges before searching the minimum element. Let's assume we have n sized array. All possible ranges for the array is approximately n^2. Ranges are [0,0], [0,1], [0,2] … [0,n] (n+1 many) + [1,1], [1,2], [1,3] … [1,n] (n many) to goes until the range is just [n,n] (1 many). Therefore, all the ranges are sum up to $\frac{n(n+1)}{2} + (n+1)$ which is approximately n^2 many ranges.

2.2.1. Pseudocode

int precomputeAll [][] (int [] arr)

```
n ← arr length
int minTable [][] ← [n][n]
for int l ← 0 to l ← n do
      minTable [l][l] ← arr[l]
      for int r ← l + 1 to r ← n do
            minTable[l][r] ← min (minTable [l][r-1], arr[r])
return minTable
```

void minQueryPA (int [][] minTable, int L, int R)

```
minValue ← minTable[L][R]
```

2.2.2. Time and Space Complexity

Preprocessing for this algorithm is precomputing the minimum element for all possible ranges. There are n^2 possible ranges for array size n. These minimum values are stored in the 2-dimensional look-up array. Later we can find the minimum element directly looking at the 2-D array Therefore, the time complexity for preprocessing is:

$$T = \sum_{l=0}^{n-1} \sum_{r=l+1}^{n-1} 1 = \sum_{l=0}^{n-1} (n - l - 1)$$

$$\cong \frac{n(n-1)}{2} = O(n^2)$$

Best case time complexity (preprocessing): O(1) (array that have only one element).
Worst case time complexity (preprocessing): $O(n^2)$

After preprocessing, finding minimum element from the table for given range is in linear time O(1).

We know that there's n^2 possible range for n sized array. We need to store all these ranges. Since we use nxn 2-dimensional array for storing, the space complexity is $O(n^2)$.

2.3. Sparse Table

The sparse table method precomputes minimum values for specific power-of-two-sized ranges in the array. The sparse table focuses only on ranges of sizes 2^0, 2^1, 2^2, 2^3… 2^k, where k is $\lfloor log2(n) \rfloor$.

2.3.1. Pseudocode

void sparseTable(int [] arr)

```
    int n ← arr length
    int k ←  log(n) / (log (2) + 1)
    int sparseTable ← [n][k]
    for i ← 0 to i ← n do
        sparseTable[i][0] ← arr[i]
    for j ← 1 to (1<<j) ← n do
        for i ← 0 to i + (1<<j) – 1 ← n do
```

```
sparseTable[i][j] ← min (sparseTable[i][j - 1],
                         sparseTable[i + (1 << (j - 1))][j - 1])
```

```
void minQueryST(int L, int R)
    int j ← log (R – L + 1) / log (2)
    int minValue ← min(sparseTable[L][j], sparseTable[R - (1 << j) + 1][j])
```

2.3.2. Time and Space Complexity

Preprocessing for the sparse table is precomputing all answers for range queries with power of length two. Afterwards, a different range query can be answered by splitting the range into ranges with power of two lengths, looking up the precomputed answers, and combining them to receive a complete answer. Therefore, the time complexity of sparse table can compute as:

$$
\begin{aligned}
T = \sum_{j=1}^{\log 2(n)} \sum_{i=1}^{n-2^j} 1 = \sum_{j=1}^{\log 2(n)} n - 2^j + 1 \\
= n\lfloor \log 2(n) \rfloor - (2^{\lfloor \log 2(n) \rfloor + 1} - 2) + \lfloor \log 2(n) \rfloor \\
\cong O(nlog(n))
\end{aligned}
$$

Best case time complexity: O(nlog(n))

Worst case time complexity: O(nlog(n))

After preprocessing, finding minimum element from the table for given range is in linear time O(1).

We used a 2-dimensional array for storing the answers to the precomputed queries. [i][j] will store the answer for the range [j, j + 2i -1] of length 2^i. The size of the 2-dimensional array is kxn where n is the array size and k is the power of two range according to the n. k must satisfy k >= log2(n). Therefore, the space complexity is kxn due to the table which is approximately O(nlog(n)).

2.4. Blocking Decomposition

Blocking decomposition is a method for efficiently answering range minimum queries by dividing the array into fixed-size blocks. Within each block, the minimum element is precomputed and stored, allowing quick access during queries. When processing a query, the minimum is determined by combining the precomputed block minimums and performing a linear scan on the partial blocks at the start and end of the range.

2.4.1. Pseudocode

```
void blocking (int [] arr, int bSize)
        int blockSize ← bSize
        int n ← arr length
        int numBlocks ← (n + blockSize -1) / blockSize
        int blockMins [] = [numBlocks]
        array fill blockMins ← int MAX
        for i ← 0 to i ← n do
                int blockIndex ← i / blockSize
                blockMins[blockIndex] ← min(blockMins[blockIndex], arr[i])

void minQueryBlock(int[] arr, int L, int R)
>       int minValue ← int MAX
        int startBlock ← L / blockSize
        int enBlock ← R / blockSize
        if startBlock == endBlock then
                for i ← L to i ← R do
                        minValue ← min (minValue, arr[i])
        else then
                for i ← L to i ← (startBlock + 1) * blockSize and i < arr length do
                        min (minValue, arr[i])
                for block ← startBlock+1 to block < endBlock
                                                and block < blockMins length do
                        min (minValue, blockMins[block])
                for i ← endBlock * blockSize to i ← R and i ← arr length do
                        min (minValue, arr[i])
```

2.4.2. Time and Space Complexity

Preprocessing for the blocking decomposition is dividing the array range by determined block size n. After blocking the array, the algorithm computes the minimum for each block by comparing the elements in the block. Each block minimum stored in the total block count sized array. Let's say block size is equal to some integer b, then block count roughly equals to n/b (n is the input array length). So the time complexity for the preprocessing is roughly equal to formula in the below:

$$T = \sum_{i=0}^{b} \frac{n}{b} \cong O(n)$$

Best case time complexity (preprocessing): O(1) (n = b = 1)

Worst case time complexity (preprocessing): O(n)

After preprocessing, the querying is in O(b + n/b) time.

Optimizing b: for minimizing b + n/b, let's start by taking derivative.

$$\frac{d}{db}\left(b + \frac{n}{b}\right) = 1 - \frac{n}{b^2}$$

Now we can set the derivative to zero:

$$1 - \frac{n}{b^2} = 0$$

$$1 = \frac{n}{b^2}$$

$$b^2 = n$$

$$b = \sqrt{n}$$

So, for the best query process, we should choose b as square root of input array size.

Since we need to use another array then input array, algorithm is not in place. So the space complexity is O($\lceil n/b \rceil$) (for number of blocks). And choosing b as $\sqrt{n}$ makes the space complexity O($\sqrt{n}$).

3. Experimental Design

The code tests and compares different methods for solving the Range Minimum Query (RMQ) problem. It looks at four approaches: direct computation, full precomputation, sparse tables, and a blocking method. These methods differ in how much time they spend preparing data (preprocessing) versus answering queries.

The experiment uses datasets of different sizes (100 to 12,000 elements) and types, like random numbers, sorted arrays, and reverse-sorted arrays. Each method first preprocesses the data, and the time for this step is measured. After that, the code runs queries to find the minimum value in a range and records how long the queries take.

By testing these methods on various datasets, the experiment shows how the size and type of the data affect preprocessing and query speed. It helps understand the trade-offs between faster setup time and quicker queries, so you can pick the best method depending on your needs.

4. Results

4.1. Precompute None

```
minimum of range [0, 99] is: 1
precompute none total query time: 74000 ns
minimum of range [0, 299] is: 0
precompute none total query time: 185100 ns
minimum of range [0, 499] is: 10
precompute none total query time: 41600 ns
minimum of range [0, 999] is: 0
precompute none total query time: 77500 ns
minimum of range [0, 2999] is: 2
precompute none total query time: 120100 ns
minimum of range [0, 4999] is: 2
precompute none total query time: 112000 ns
minimum of range [0, 9999] is: 0
precompute none total query time: 209500 ns
minimum of range [0, 11999] is: 2
precompute none total query time: 499100 ns
```

Figure 1: random arrays


```
minimum of range [0, 99] is: 1
precompute none total query time: 32500 ns
minimum of range [0, 299] is: 0
precompute none total query time: 335100 ns
minimum of range [0, 499] is: 10
precompute none total query time: 250600 ns
minimum of range [0, 999] is: 0
precompute none total query time: 49000 ns
minimum of range [0, 2999] is: 2
precompute none total query time: 74000 ns
minimum of range [0, 4999] is: 2
precompute none total query time: 121000 ns
minimum of range [0, 9999] is: 0
precompute none total query time: 192200 ns
minimum of range [0, 11999] is: 2
precompute none total query time: 327200 ns
```

Figure 2: sorted arrays

```
minimum of range [0, 99] is: 1
precompute none total query time: 139200 ns
minimum of range [0, 299] is: 0
precompute none total query time: 37400 ns
minimum of range [0, 499] is: 10
precompute none total query time: 39100 ns
minimum of range [0, 999] is: 0
precompute none total query time: 69300 ns
minimum of range [0, 2999] is: 2
precompute none total query time: 105300 ns
minimum of range [0, 4999] is: 2
precompute none total query time: 220400 ns
minimum of range [0, 9999] is: 0
precompute none total query time: 210800 ns
minimum of range [0, 11999] is: 2
precompute none total query time: 325200 ns
```

Figure 3: reverse sorted arrays

4.2. Precompute All

```
precompute all preproccesing time: 355200 ns
minimum of range [0, 99] is: 1
precompute all total Query Time: 27000 ns

precompute all preproccesing time: 1470100 ns
minimum of range [0, 299] is: 0
precompute all total Query Time: 25700 ns

precompute all preproccesing time: 1772600 ns
minimum of range [0, 499] is: 10
precompute all total Query Time: 28600 ns

precompute all preproccesing time: 3956700 ns
minimum of range [0, 999] is: 0
precompute all total Query Time: 26900 ns
```

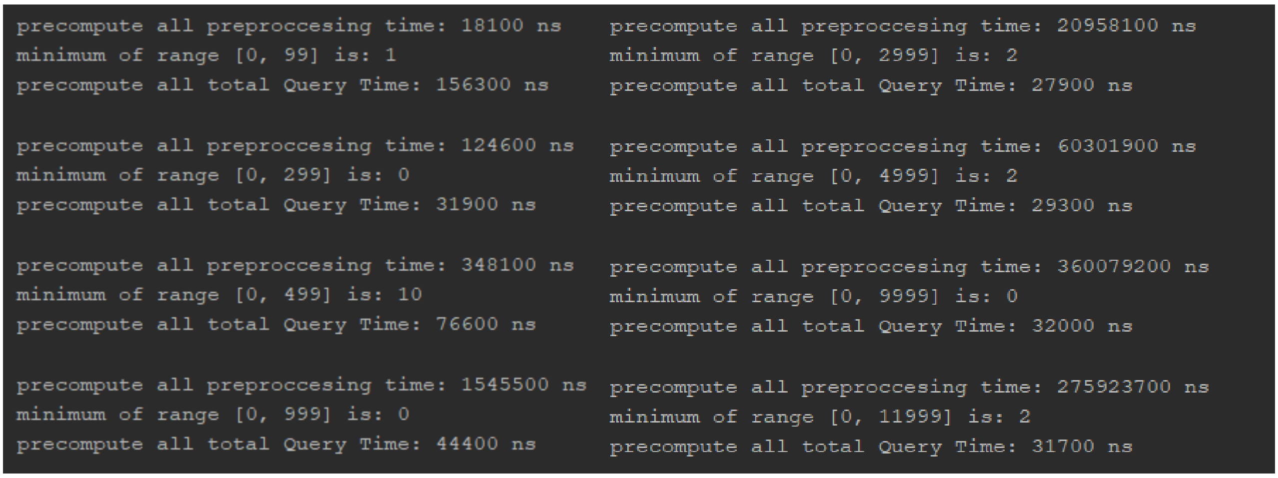
```
precompute all preproccesing time: 24131200 ns
minimum of range [0, 2999] is: 2
precompute all total Query Time: 40800 ns

precompute all preproccesing time: 69955000 ns
minimum of range [0, 4999] is: 2
precompute all total Query Time: 36800 ns

precompute all preproccesing time: 296172900 ns
minimum of range [0, 9999] is: 0
precompute all total Query Time: 61300 ns

precompute all preproccesing time: 315116000 ns
minimum of range [0, 11999] is: 2
precompute all total Query Time: 33400 ns
```


Figure 4 and 5: random arrays

```
precompute all preproccesing time: 494700 ns
minimum of range [0, 99] is: 1
precompute all total Query Time: 39600 ns

precompute all preproccesing time: 372600 ns
minimum of range [0, 299] is: 0
precompute all total Query Time: 30500 ns

precompute all preproccesing time: 702100 ns
minimum of range [0, 499] is: 10
precompute all total Query Time: 70200 ns

precompute all preproccesing time: 2387900 ns
minimum of range [0, 999] is: 0
precompute all total Query Time: 58600 ns
```

```
precompute all preproccesing time: 22081200 ns
minimum of range [0, 2999] is: 2
precompute all total Query Time: 67600 ns

precompute all preproccesing time: 56129700 ns
minimum of range [0, 4999] is: 2
precompute all total Query Time: 31200 ns

precompute all preproccesing time: 278601100 ns
minimum of range [0, 9999] is: 0
precompute all total Query Time: 88100 ns

precompute all preproccesing time: 332961800 ns
minimum of range [0, 11999] is: 2
precompute all total Query Time: 32900 ns
```

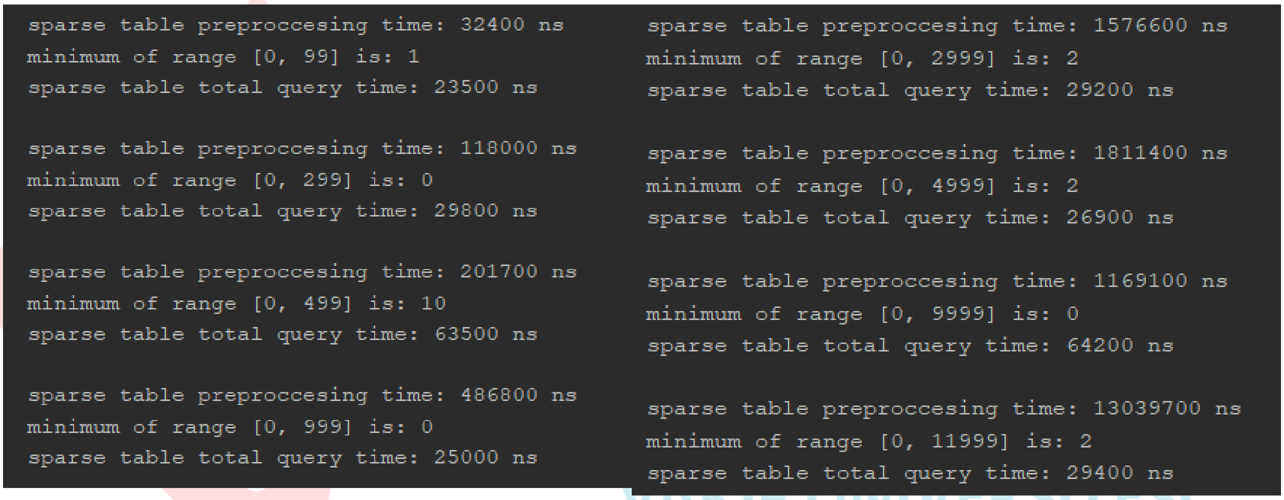
Figure 6 and 7: sorted arrays



```
precompute all preproccesing time: 18100 ns
minimum of range [0, 99] is: 1
precompute all total Query Time: 156300 ns

precompute all preproccesing time: 124600 ns
minimum of range [0, 299] is: 0
precompute all total Query Time: 31900 ns

precompute all preproccesing time: 348100 ns
minimum of range [0, 499] is: 10
precompute all total Query Time: 76600 ns

precompute all preproccesing time: 1545500 ns
minimum of range [0, 999] is: 0
precompute all total Query Time: 44400 ns

precompute all preproccesing time: 20958100 ns
minimum of range [0, 2999] is: 2
precompute all total Query Time: 27900 ns

precompute all preproccesing time: 60301900 ns
minimum of range [0, 4999] is: 2
precompute all total Query Time: 29300 ns

precompute all preproccesing time: 360079200 ns
minimum of range [0, 9999] is: 0
precompute all total Query Time: 32000 ns

precompute all preproccesing time: 275923700 ns
minimum of range [0, 11999] is: 2
precompute all total Query Time: 31700 ns
```

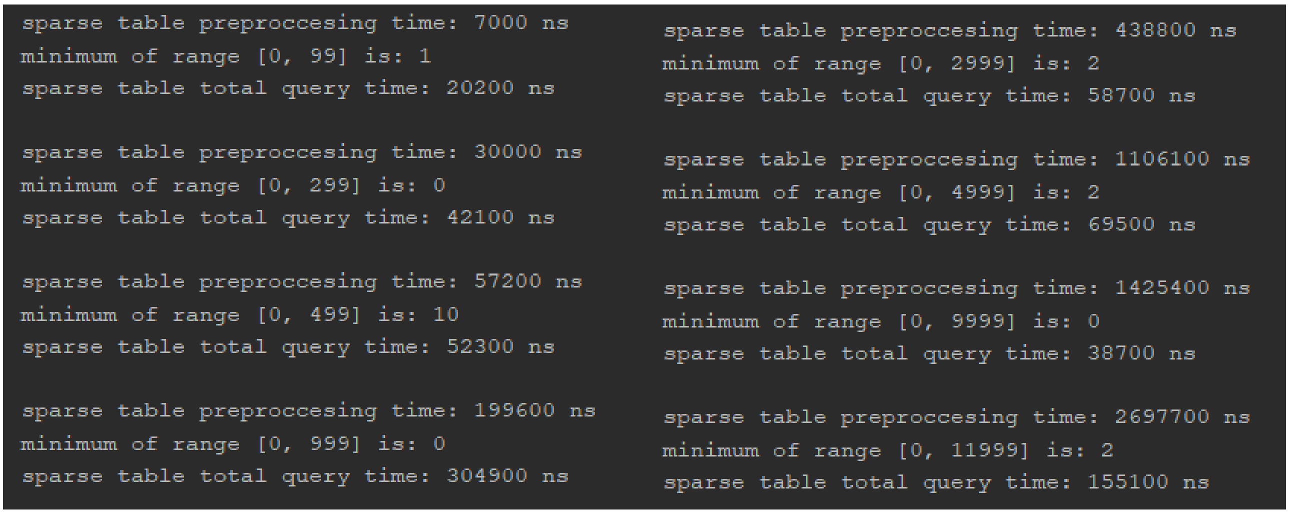
Figure 8 and 9: reverse sorted arrays

4.3. Sparse Table



```
sparse table preproccesing time: 32400 ns
minimum of range [0, 99] is: 1
sparse table total query time: 23500 ns

sparse table preproccesing time: 118000 ns
minimum of range [0, 299] is: 0
sparse table total query time: 29800 ns

sparse table preproccesing time: 201700 ns
minimum of range [0, 499] is: 10
sparse table total query time: 63500 ns

sparse table preproccesing time: 486800 ns
minimum of range [0, 999] is: 0
sparse table total query time: 25000 ns

sparse table preproccesing time: 1576600 ns
minimum of range [0, 2999] is: 2
sparse table total query time: 29200 ns

sparse table preproccesing time: 1811400 ns
minimum of range [0, 4999] is: 2
sparse table total query time: 26900 ns

sparse table preproccesing time: 1169100 ns
minimum of range [0, 9999] is: 0
sparse table total query time: 64200 ns

sparse table preproccesing time: 13039700 ns
minimum of range [0, 11999] is: 2
sparse table total query time: 29400 ns
```

Figure 10 and 11: random arrays



```
sparse table preproccesing time: 7000 ns
minimum of range [0, 99] is: 1
sparse table total query time: 20200 ns

sparse table preproccesing time: 30000 ns
minimum of range [0, 299] is: 0
sparse table total query time: 42100 ns

sparse table preproccesing time: 57200 ns
minimum of range [0, 499] is: 10
sparse table total query time: 52300 ns

sparse table preproccesing time: 199600 ns
minimum of range [0, 999] is: 0
sparse table total query time: 304900 ns

sparse table preproccesing time: 438800 ns
minimum of range [0, 2999] is: 2
sparse table total query time: 58700 ns

sparse table preproccesing time: 1106100 ns
minimum of range [0, 4999] is: 2
sparse table total query time: 69500 ns

sparse table preproccesing time: 1425400 ns
minimum of range [0, 9999] is: 0
sparse table total query time: 38700 ns

sparse table preproccesing time: 2697700 ns
minimum of range [0, 11999] is: 2
sparse table total query time: 155100 ns
```

Figure 12 and 13: sorted arrays

```
sparse table preproccesing time: 11200 ns
minimum of range [0, 99] is: 1
sparse table total query time: 68200 ns

sparse table preproccesing time: 48200 ns
minimum of range [0, 299] is: 0
sparse table total query time: 35700 ns

sparse table preproccesing time: 60100 ns
minimum of range [0, 499] is: 10
sparse table total query time: 32700 ns

sparse table preproccesing time: 154000 ns
minimum of range [0, 999] is: 0
sparse table total query time: 31600 ns
```

```
sparse table preproccesing time: 542300 ns
minimum of range [0, 2999] is: 2
sparse table total query time: 36200 ns

sparse table preproccesing time: 866800 ns
minimum of range [0, 4999] is: 2
sparse table total query time: 35800 ns

sparse table preproccesing time: 1287500 ns
minimum of range [0, 9999] is: 0
sparse table total query time: 79500 ns

sparse table preproccesing time: 1585900 ns
minimum of range [0, 11999] is: 2
sparse table total query time: 52300 ns
```

Figure 14 and 15: reverse sorted arrays

4.4. Blocking

```
blocking preprocessing time: 10400 ns
minimum of range [0, 99] is: 1
blocking total query time: 37700 ns

blocking preprocessing time: 24700 ns
minimum of range [0, 299] is: 0
blocking total query time: 133800 ns

blocking preprocessing time: 33100 ns
minimum of range [0, 499] is: 10
blocking total query time: 46300 ns

blocking preprocessing time: 76500 ns
minimum of range [0, 999] is: 0
blocking total query time: 33500 ns
```

```
blocking preprocessing time: 192700 ns
minimum of range [0, 2999] is: 2
blocking total query time: 54700 ns

blocking preprocessing time: 309800 ns
minimum of range [0, 4999] is: 2
blocking total query time: 32300 ns

blocking preprocessing time: 470900 ns
minimum of range [0, 9999] is: 0
blocking total query time: 47800 ns

blocking preprocessing time: 447100 ns
minimum of range [0, 11999] is: 2
blocking total query time: 37600 ns
```

Figure 16 and 17: random arrays

```
blocking preprocessing time: 5500 ns
minimum of range [0, 99] is: 1
blocking total query time: 18700 ns

blocking preprocessing time: 12700 ns
minimum of range [0, 299] is: 0
blocking total query time: 37200 ns

blocking preprocessing time: 21400 ns
minimum of range [0, 499] is: 10
blocking total query time: 21200 ns

blocking preprocessing time: 39100 ns
minimum of range [0, 999] is: 0
blocking total query time: 43100 ns
```

```
blocking preprocessing time: 163700 ns
minimum of range [0, 2999] is: 2
blocking total query time: 37800 ns

blocking preprocessing time: 249200 ns
minimum of range [0, 4999] is: 2
blocking total query time: 47000 ns

blocking preprocessing time: 476500 ns
minimum of range [0, 9999] is: 0
blocking total query time: 32100 ns

blocking preprocessing time: 497500 ns
minimum of range [0, 11999] is: 2
blocking total query time: 52400 ns
```

Figure 18 and 19: sorted arrays

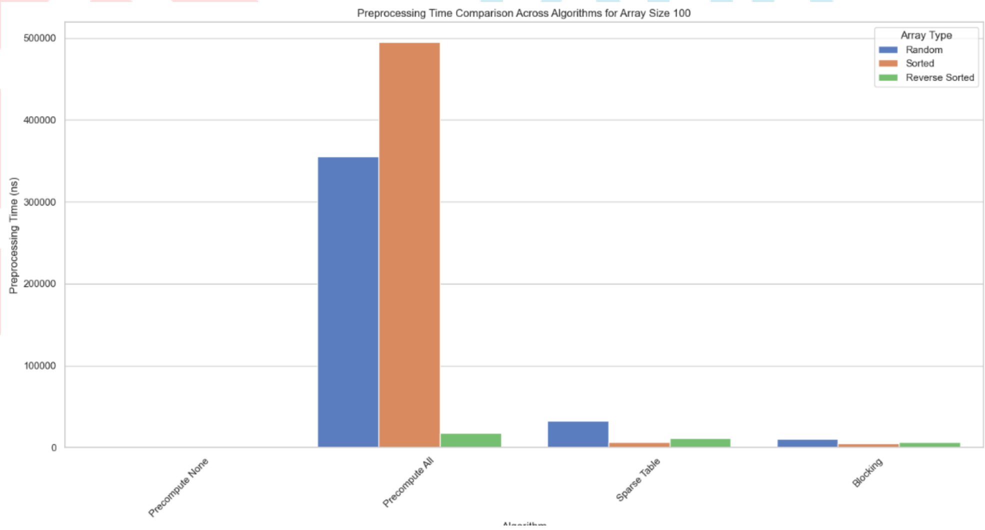
```
blocking preprocessing time: 7000 ns
minimum of range [0, 99] is: 1
blocking total query time: 17000 ns

blocking preprocessing time: 29900 ns
minimum of range [0, 299] is: 0
blocking total query time: 16300 ns

blocking preprocessing time: 20500 ns
minimum of range [0, 499] is: 10
blocking total query time: 15300 ns

blocking preprocessing time: 21900 ns
minimum of range [0, 999] is: 0
blocking total query time: 17300 ns
```

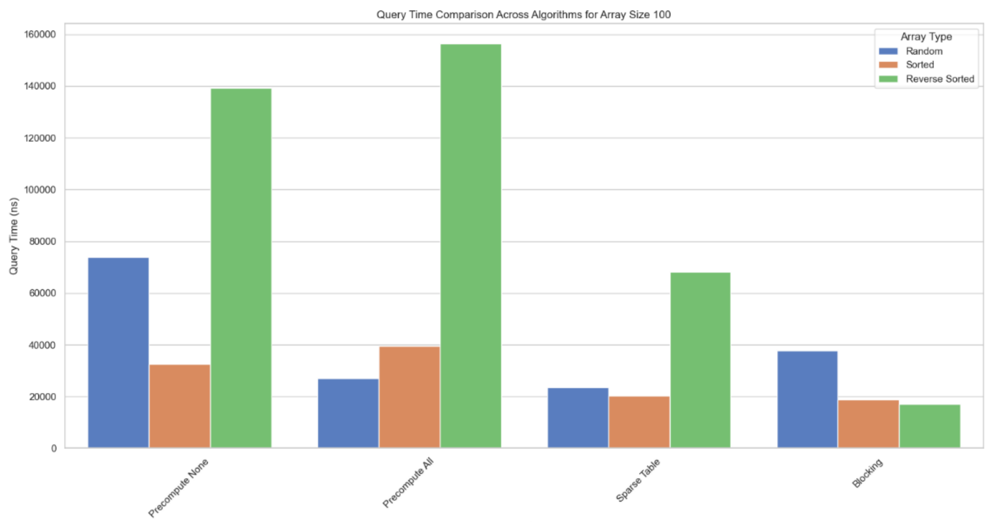
```
blocking preprocessing time: 26400 ns
minimum of range [0, 2999] is: 2
blocking total query time: 20200 ns

blocking preprocessing time: 22600 ns
minimum of range [0, 4999] is: 2
blocking total query time: 18900 ns

blocking preprocessing time: 64800 ns
minimum of range [0, 9999] is: 0
blocking total query time: 26300 ns

blocking preprocessing time: 55900 ns
minimum of range [0, 11999] is: 2
blocking total query time: 24200 ns
```

Figure 20 and 21: reverse sorted arrays

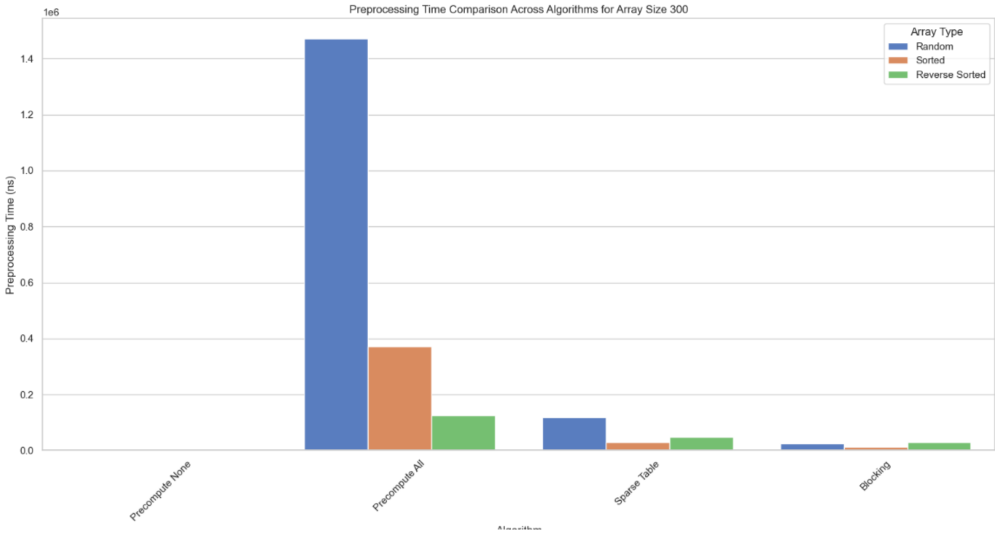
4.5. Graphs

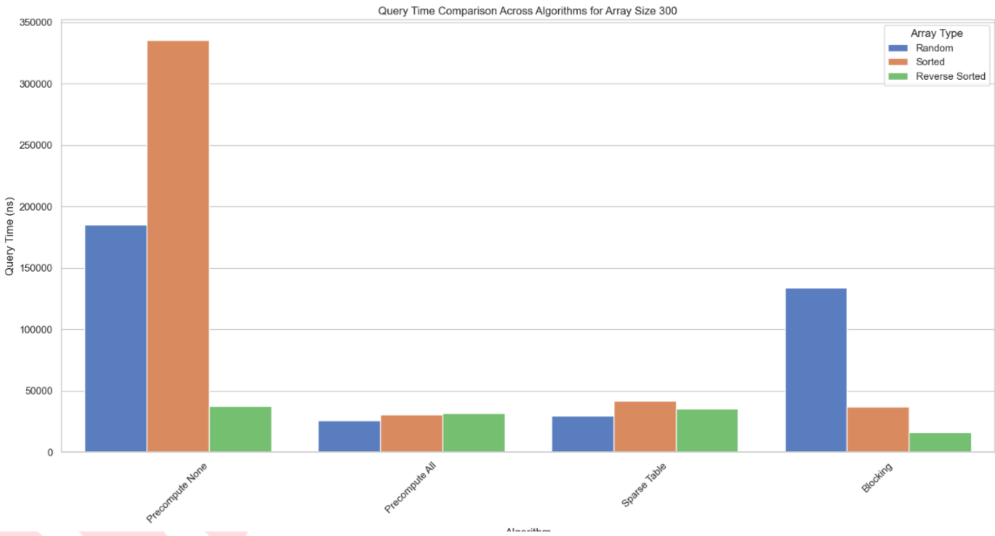
4.5.1. Array Size 100



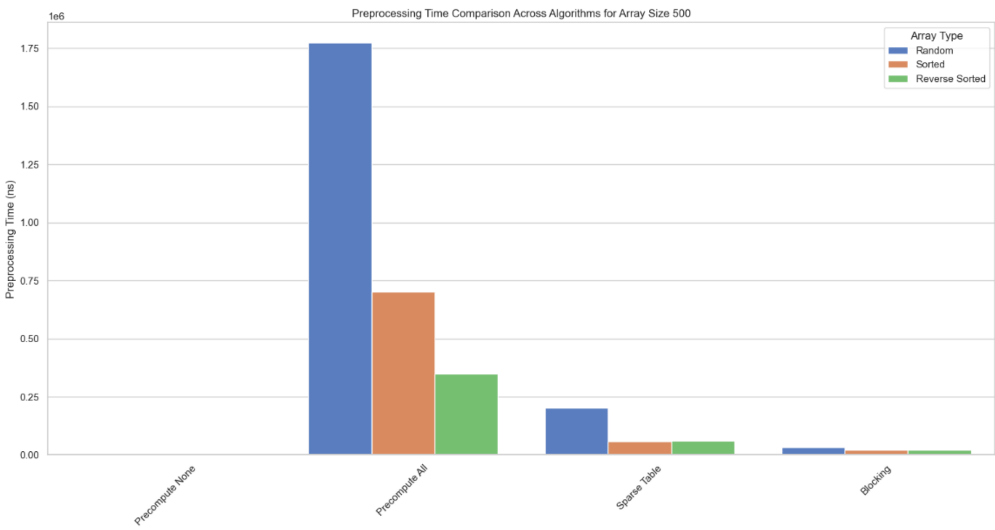


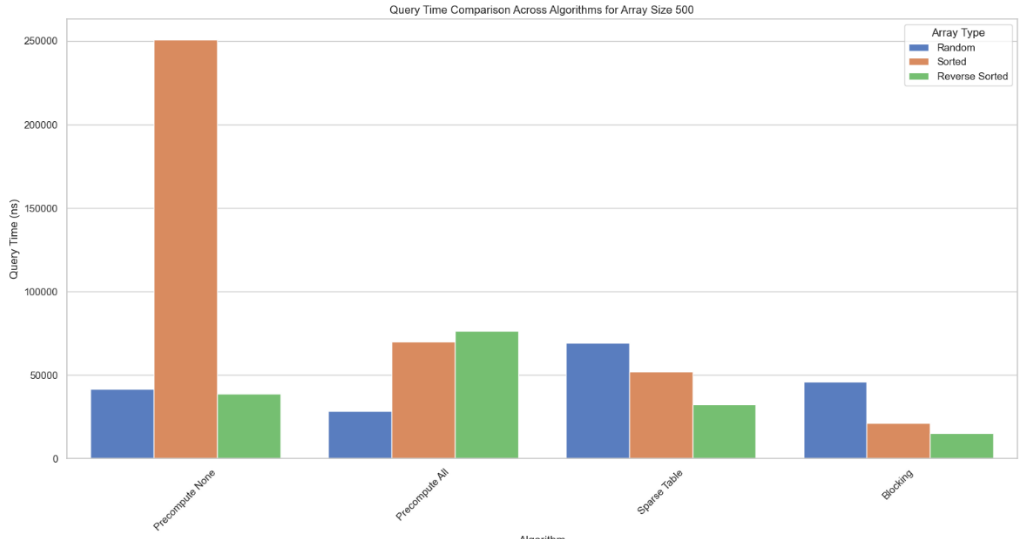
4.5.2. Array Size 300



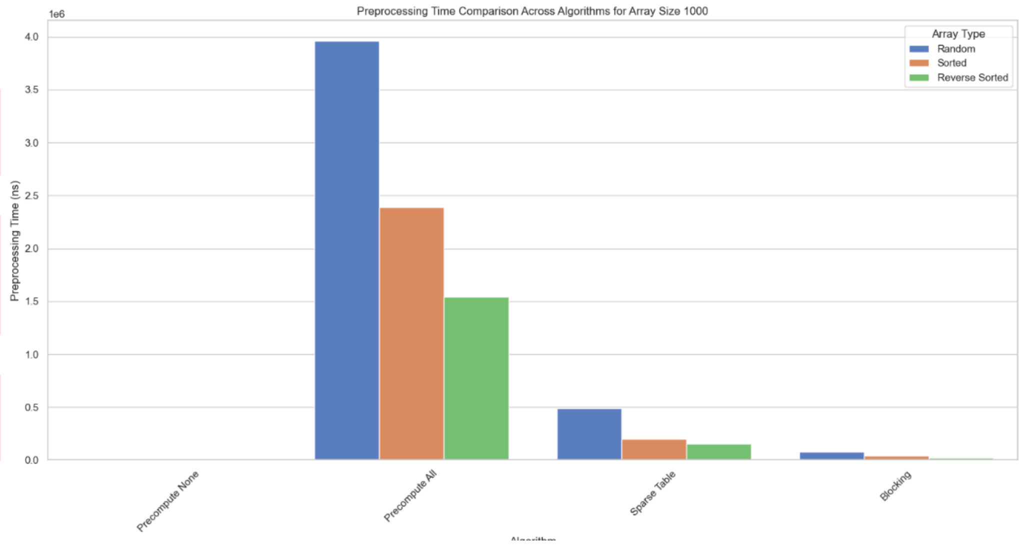


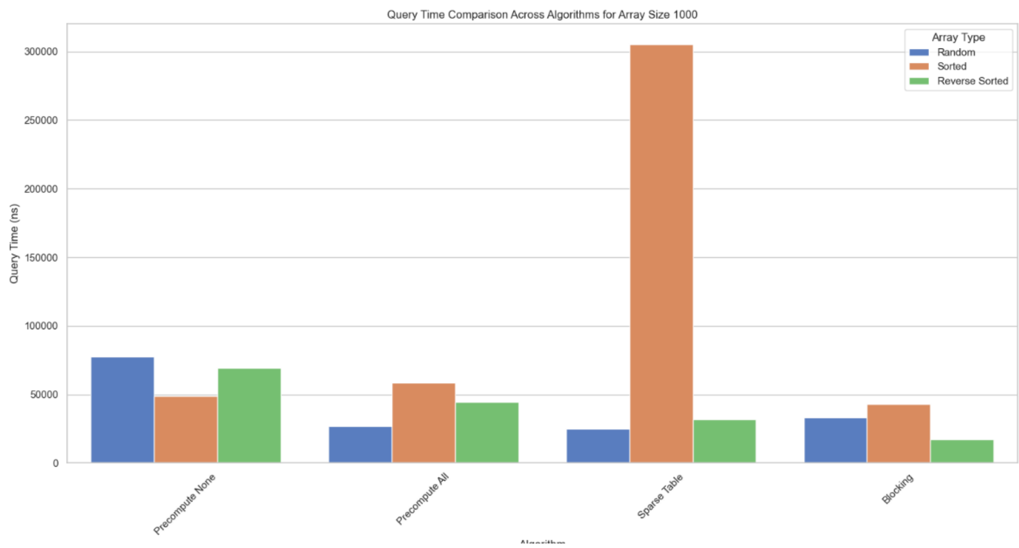
4.5.3. Array Size 500



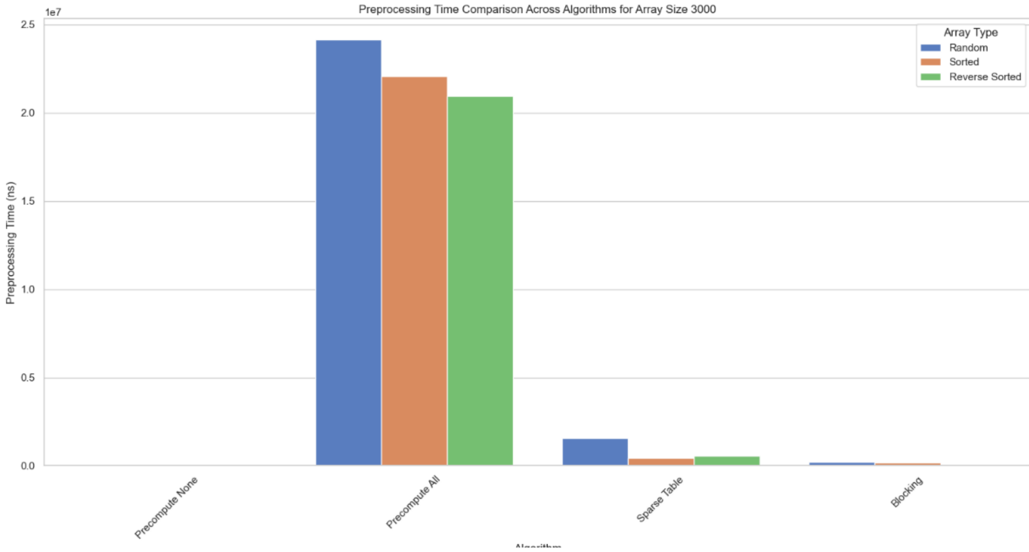


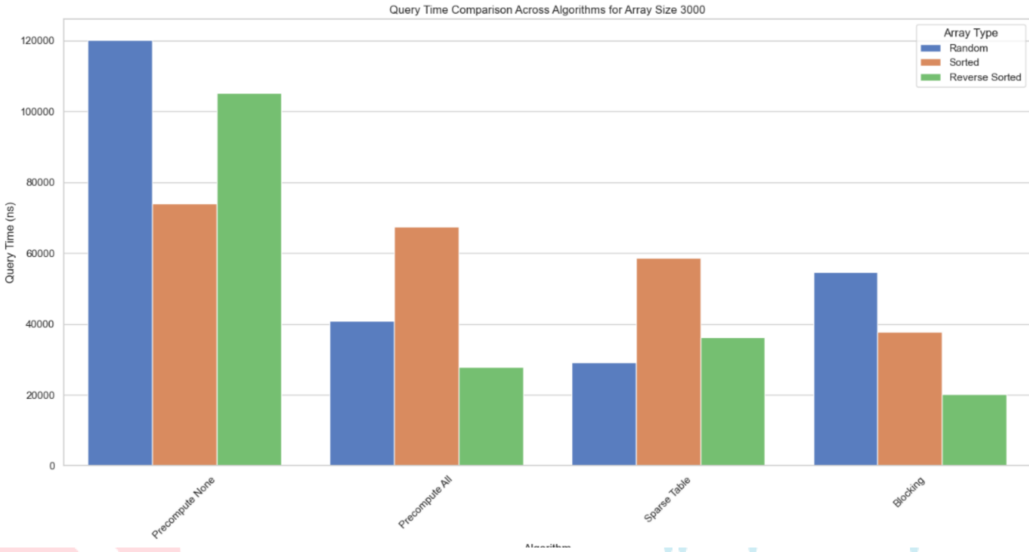
4.5.4. Array Size 1000



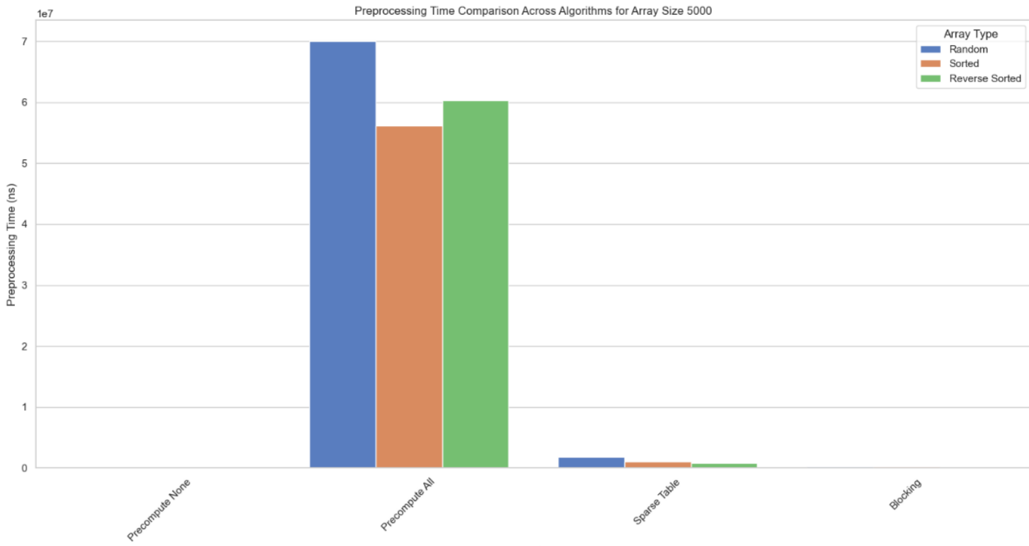


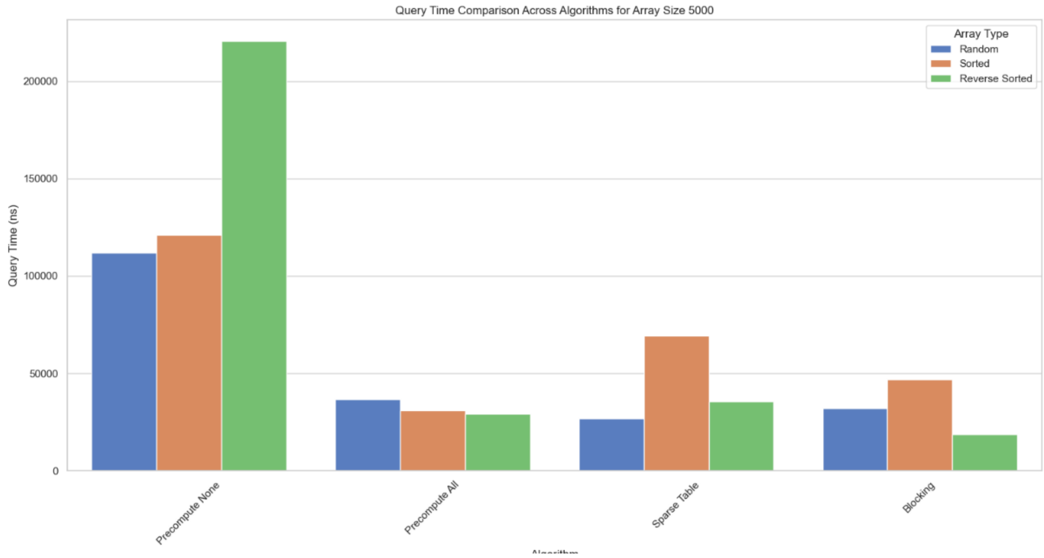
4.5.5. Array Size 3000



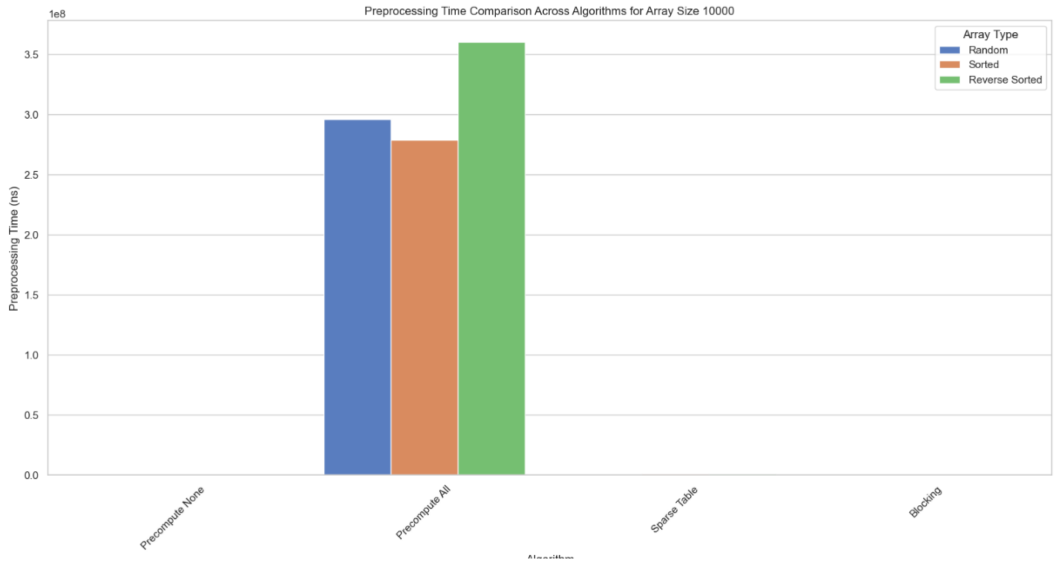


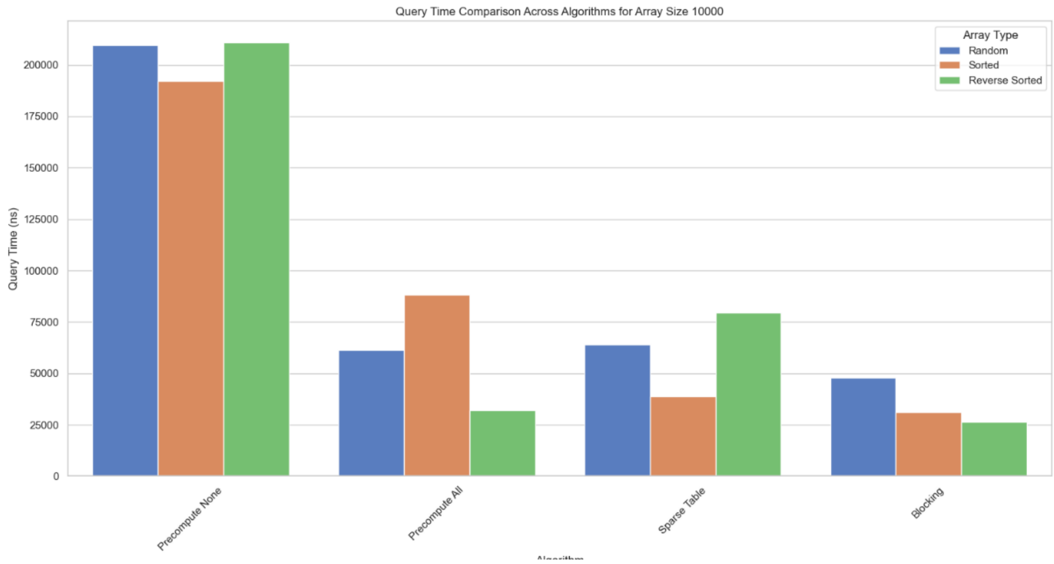
4.5.6. Array Size 5000



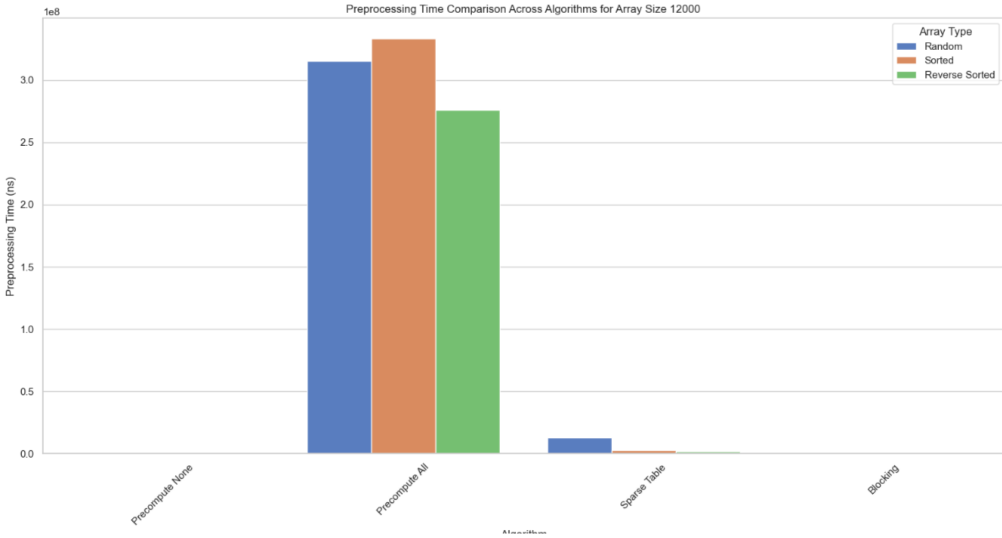


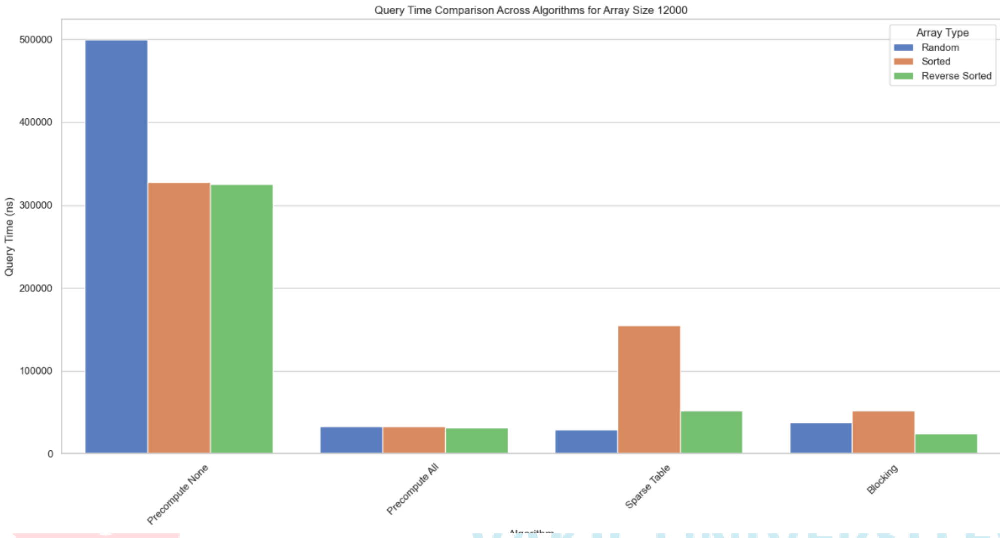
4.5.7. Array Size 10000



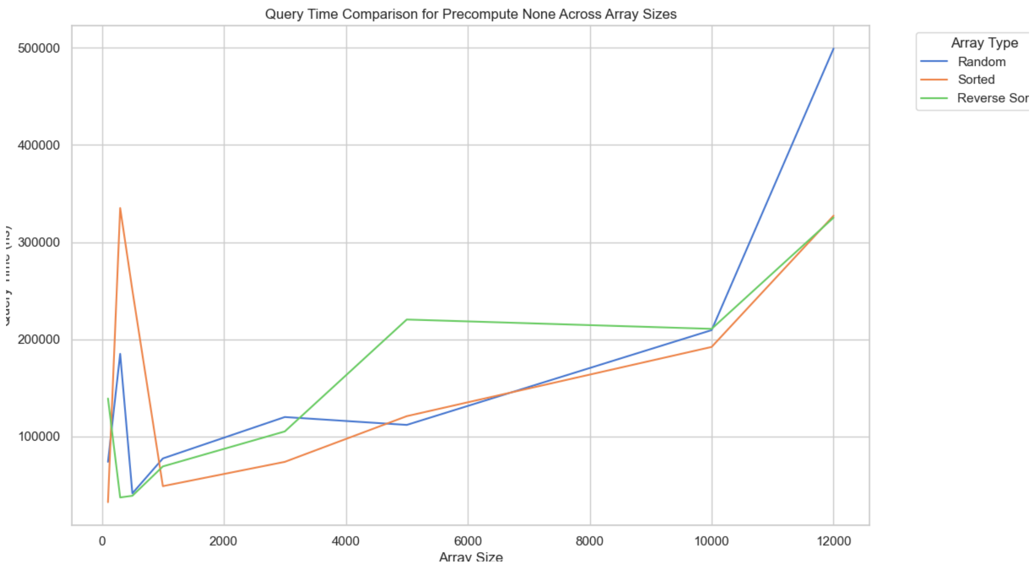


4.5.8. Array Size 12000

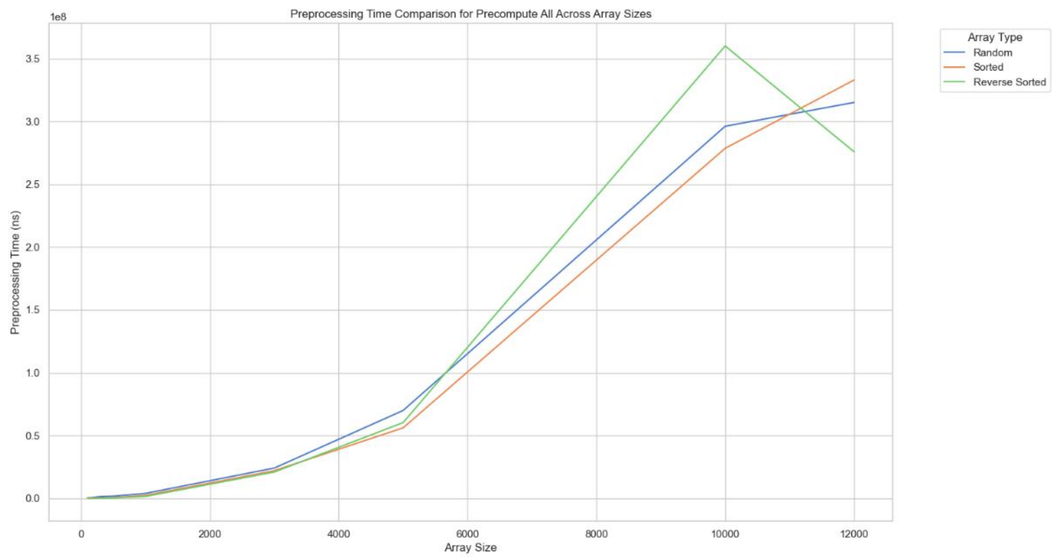


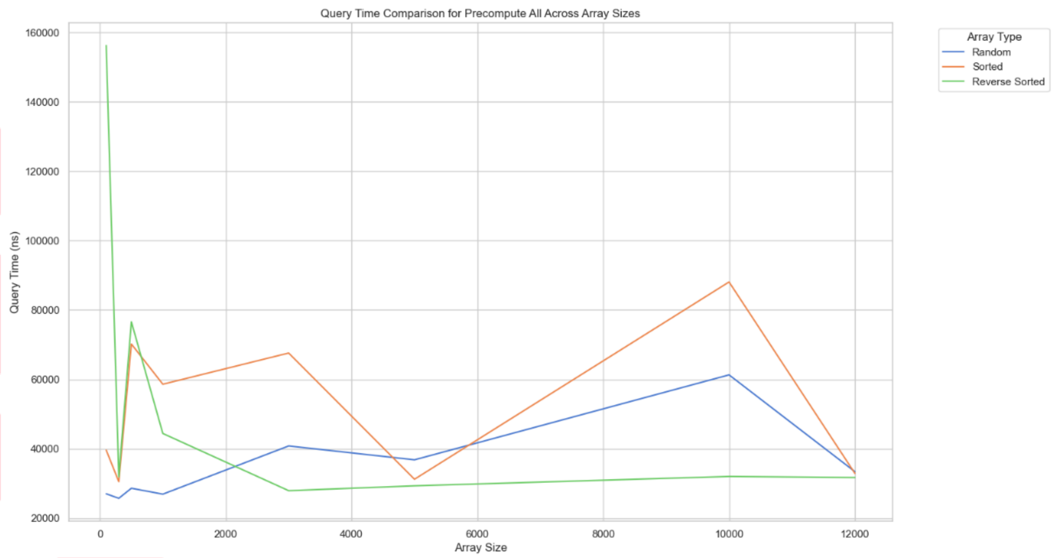


4.5.9. Precompute None

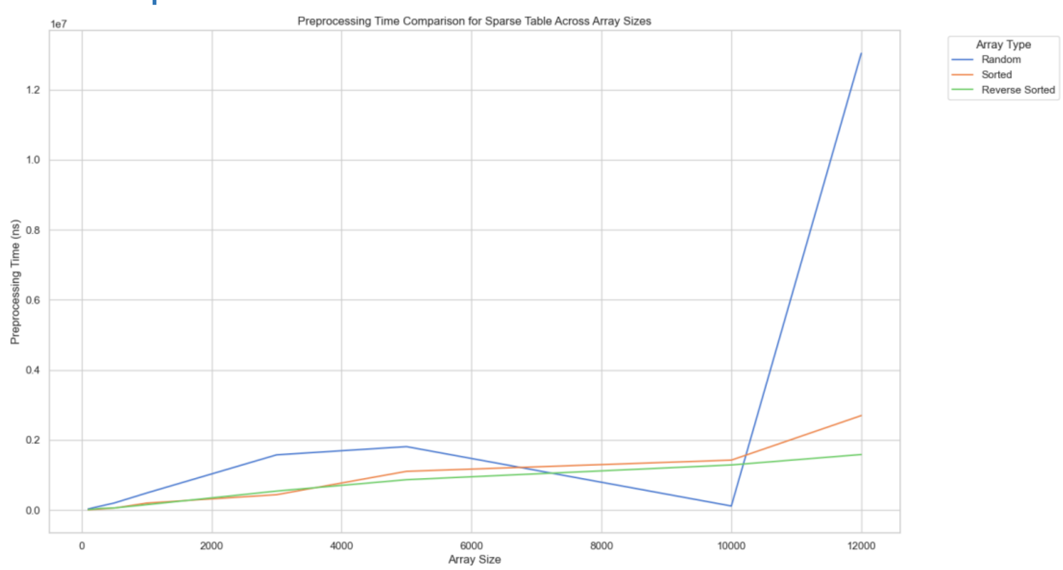


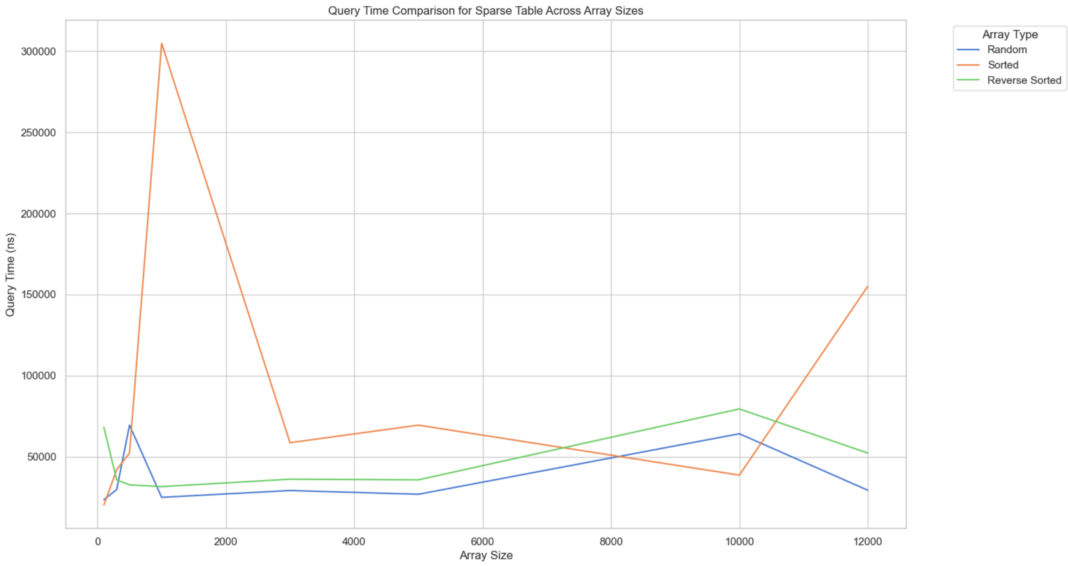
4.5.10. Precompute All



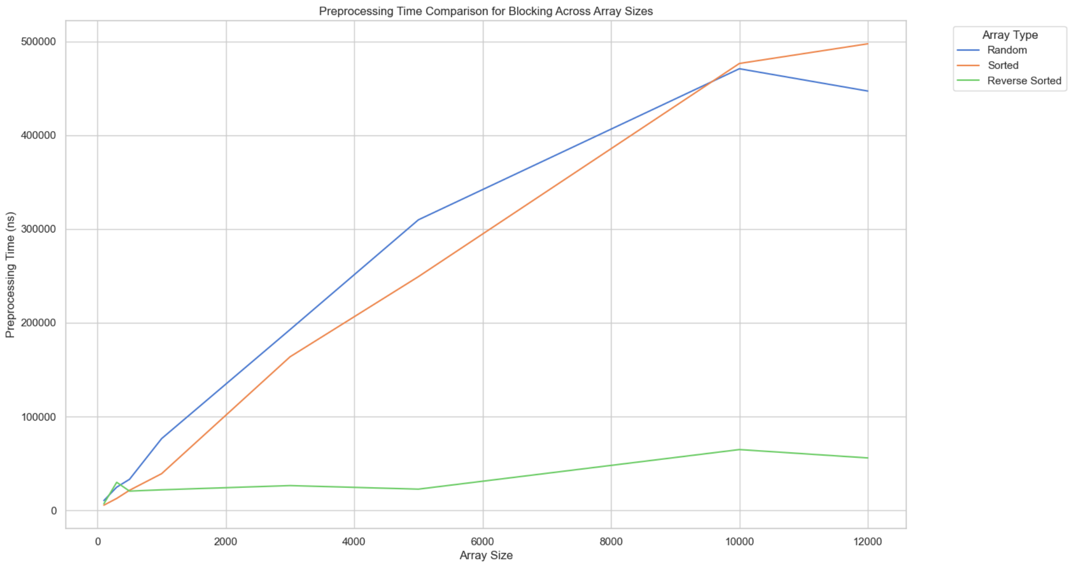


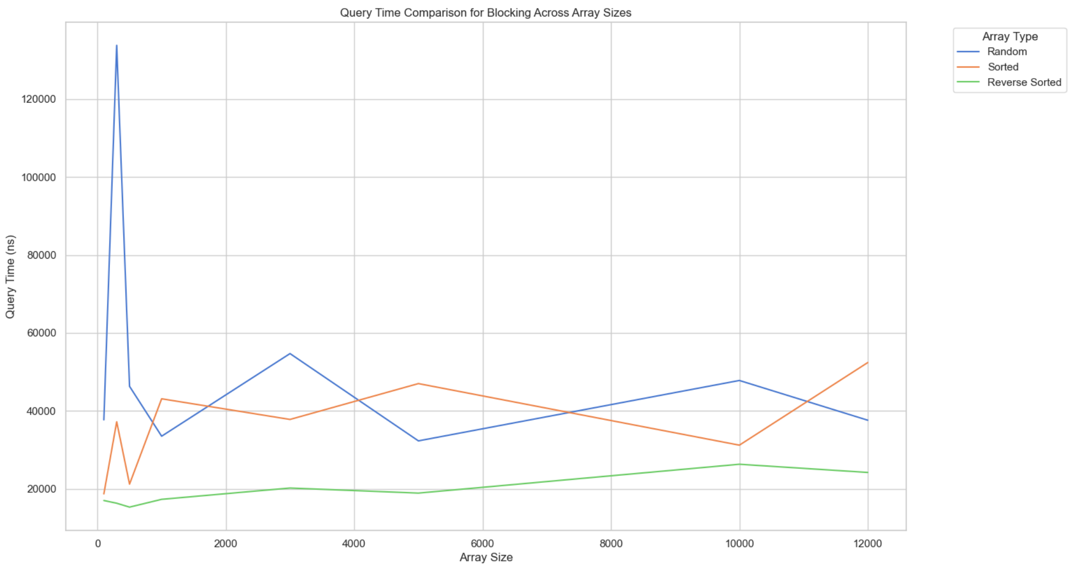
4.5.11. Sparse Table





4.5.12. Block Decomposition





5. Conclusion

In this study, we looked at four ways to solve the Range Minimum Query (RMQ) problem: precompute none, precompute all, sparse table, and blocking decomposition. Each method has its own strengths and weaknesses in terms of speed, memory use, and preprocessing time.

Precompute none is the simplest algorithm as it does not require any preprocessing. This makes it suitable for small datasets or situations where the number of queries is minimal. However, its linear query time makes it inefficient for larger datasets, and its performance remains consistent across sorted, random, and reverse-sorted data since it evaluates the range directly.

Precompute all represents the opposite approach by precomputing and storing the minimum values for all possible ranges. This allows queries to be answered instantly. While it performs well for sorted and reverse-sorted datasets, its preprocessing time and memory usage grow significantly for large, random datasets, limiting its practicality in resource-constrained environments.

Sparse table strikes a balance between preprocessing and query efficiency. It computes minimum values for power-of-two-sized ranges. Its performance is excellent on random and unsorted data, as it avoids the excessive memory usage associated with precompute all while still ensuring fast query times. Sparse table is particularly advantageous when preprocessing time is acceptable.

Blocking decomposition divides the dataset into fixed-size blocks and precomputes the minimum for each block. This approach reduces preprocessing time and memory requirements compared to precompute all while maintaining reasonably fast query times. It is effective for both random and sorted datasets, though its performance can be slightly slower on reverse-sorted data due to the structure of the blocks.

The results demonstrate that dataset type plays a critical role in determining the best algorithm. Sorted and reverse-sorted datasets benefit from algorithms like Precompute all, where preprocessing effort leads to instant query times. Random datasets are better suited for sparse table or blocking decomposition, which balance preprocessing and query efficiency.

Additionally, when examining the graphs, some results deviate from theoretical expectations. For example, in a low-dimensional array, results expected to be low appear unexpectedly high. However, when the overall trend in the graphs is considered, the increases generally align with the calculations. These unexpected results might stem from the system on which the algorithms were executed.

Overall, there is no universally optimal algorithm for RMQ problems. The choice depends on factors such as dataset size, type, number of queries, and available resources. By understanding the trade-offs between preprocessing time, query speed, and memory usage, one can select the most suitable algorithm for a specific use case.

6. References

1. https://www.sciencedirect.com/science/article/pii/S030439752200038X
2. https://iq.opengenus.org/range-minimum-query-naive/
3. https://strncat.github.io/jekyll/update/2019/03/22/rmq.html
4. https://cp-algorithms.com/data_structures/sparse-table.html
5. https://medium.com/nybles/sparse-table-f3981fbb1bc8#:~:text=The%20main%20idea%20behind%20Sparse,to%20receive%20a%20complete%20answer.
6. https://www.geeksforgeeks.org/square-root-sqrt-decomposition-algorithm/
7. https://www.geeksforgeeks.org/sparse-table/
8. https://web.stanford.edu/class/cs166/lectures/00/Slides00.pdf
9. https://medium.com/@florian_algo/plain-and-simple-explanation-of-square-root-decomposition-cce43d8e6936
10. https://en.wikipedia.org/wiki/Range_minimum_query
11. https://cp-algorithms.com/data_structures/sqrt_decomposition.html
12. https://codeforces.com/blog/entry/78931
13. https://www.geeksforgeeks.org/range-minimum-query-for-static-array/