FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

Student:

Name: Murat

Surname: Keskin

ID Number: 2121251002

Department: Software Engineering

Project:

Topic: Range Minimum Queries (RMQ): Algorithms and Experimental Analysis

Course:

Name: SEN22311E

Instructor: Assoc. Prof. Dr. Berna Kiraz

Content

1- Problem Description

Range Minimum Query (RMQ) problem involves finding the minimum value within a specified range of a given array. This problem is used in various fields such as computer science, data structures, and algorithms, with various applications.(1)

2- Descriptions and Theoretical Analysis of Algorithms

1. Precompute All (Full Table Approach)

Method:

The Precompute All approach involves preprocessing the array to compute and store the minimum values for every possible subarray. A lookup table is created that contains precomputed minimum values for all possible ranges. This ensures that any future query can be answered immediately by referring to the table.

Complexity Analysis:

- **Preprocessing Time:** Quadratic – The time required increases significantly with the size of the array because all possible subarrays must be evaluated.
- **Query Time:** Constant – Queries are answered instantly by accessing precomputed results.
- **Space Complexity:** Quadratic – A large amount of memory is needed to store the minimum values for all subarrays.

Advantages:

- Query responses are instantaneous.
- Ideal for situations where queries are frequent, and preprocessing time is not a concern.

Disadvantages:

- High memory consumption limits its use for large datasets.
- Impractical for scenarios where the array is large, but queries are infrequent.

Pseudocode For PreCompute All:

```
function preprocessAll(A, n):
    create table minTable[n][n]
    for i from 0 to n-1:
        minTable[i][i] = A[i]  # Diagonal initialization
        for j from i+1 to n-1:
            minTable[i][j] = min(minTable[i][j-1], A[j])
```

```
function RMQ(i, j):
    return minTable[i][j]
```

2. Sparse Table Approach

Method:

The Sparse Table method is a more space-efficient alternative to the full table approach. It preprocesses the array by dividing it into overlapping segments of increasing lengths, storing minimum values for each segment. Queries are answered by combining the results from two overlapping segments that cover the specified subarray.

Complexity Analysis:

- **Preprocessing Time:** Linear logarithmic – The preprocessing phase involves computing minimums for segments of various sizes, making it more efficient than the full table method.
- **Query Time:** Constant – Queries are resolved instantly by comparing overlapping segments.
- **Space Complexity:** Linear logarithmic – The memory required grows slowly as the size of the array increases.

Advantages:

- Efficient preprocessing and minimal memory usage.
- Suitable for large datasets with frequent queries.

Disadvantages:

- Preprocessing is more complex compared to simpler methods.
- While memory requirements are lower than the full table approach, they may still be substantial for extremely large arrays.

Pseudocode For Sparse Table:

```
function preprocessSparseTable(A, n):
    logN = floor(log2(n)) + 1
    create table sparseTable[n][logN]
    for i from 0 to n-1:
        sparseTable[i][0] = A[i]  # Interval of length 1
    for j from 1 to logN:
        for i from 0 to n - 2^j + 1:
            sparseTable[i][j] = min(sparseTable[i][j-1], sparseTable[i + 2^(j-1)][j-1])
function RMQ(i, j):
    k = floor(log2(j - i + 1))
```

```
return min(sparseTable[i][k], sparseTable[j - 2^k + 1][k])
```

3. Blocking Approach (Divide and Conquer)

Method:

In the Blocking approach, the array is divided into smaller, fixed-size blocks. Minimum values are computed and stored for each block. When a query is performed, the algorithm computes the minimum by comparing results from fully covered blocks and manually evaluating the remaining elements that do not fit into a full block.

Complexity Analysis:

- **Preprocessing Time:** Linear – Preprocessing is straightforward and involves computing minimums for small, fixed-size blocks.
- **Query Time:** Square root – Query efficiency depends on the size of the blocks, resulting in slower performance compared to table-based methods.
- **Space Complexity:** Square root – Memory usage is minimal, as only block minimums are stored.

Advantages:

- Efficient preprocessing with minimal overhead.
- Balanced performance for both preprocessing and querying.
- Memory-efficient for large arrays.

Disadvantages:

- Query times are slower than methods with constant-time queries.
- Performance may degrade if block sizes are not chosen optimally.

Pseudocode For Blocking:

```
function preprocessBlocking(A, n):
    blockSize = floor(sqrt(n))
    numBlocks = ceil(n / blockSize)
    create array blockMin[numBlocks]

    for i from 0 to numBlocks-1:
        blockMin[i] = ∞
        for j from i*blockSize to min((i+1)*blockSize - 1, n-1):
            blockMin[i] = min(blockMin[i], A[j])
```

```
function RMQ(i, j):
    minVal = ∞
    while i <= j and i % blockSize != 0 and i != 0:  # Traverse left over in first block
        minVal = min(minVal, A[i])
        i += 1

    while i + blockSize - 1 <= j:  # Traverse complete blocks
        minVal = min(minVal, blockMin[i / blockSize])
        i += blockSize

    while i <= j:  # Traverse remaining elements in the last block
        minVal = min(minVal, A[i])
        i += 1

    return minVal
```

4. Precompute None (Brute-force Approach)

Method:

The Precompute None approach involves no preprocessing. Each query is answered by traversing the specified subarray and determining the minimum on the fly. This method is straightforward but inefficient for large datasets or frequent queries.

Complexity Analysis:

- **Preprocessing Time:** None – The array is not processed in advance.
- **Query Time:** Linear – Each query requires traversing the subarray, resulting in slow performance for large arrays.
- **Space Complexity:** Minimal – No additional memory is needed.

Advantages:

- Simple and easy to implement.
- Minimal memory usage.

Disadvantages:

- Extremely inefficient for large datasets or when queries are frequent.
- Query performance is poor compared to other approaches.

Pseudocode For PreCompute None:

```
function RMQ(A, i, j):
    minVal = ∞
    for k from i to j:
        minVal = min(minVal, A[k])
    return minVal
```

Comparative Analysis

Approach	Preprocessing Time	Query Time	Space Complexity	Best Use Case
Precompute All	Quadratic	Constant	Quadratic	Small arrays with numerous queries
Sparse Table	Linear logarithmic	Constant	Linear logarithmic	Large arrays with frequent queries
Blocking	Linear	Square root	Square root	Moderate-sized arrays, balanced usage
Precompute None	None	Linear	Minimal	Small datasets or rare queries

3- Results

1. Fixed Array Size

```
=== [Fixed Array Size: n=1000] ===

--- [Fixed n=1000, Testing queries=100] ---
[PrecomputeAll] n=1000, q=100 | build=0.004698 sec, query=0.000012 sec
[SparseTable] n=1000, q=100 | build=0.000134 sec, query=0.000007 sec
[Blocking] n=1000, q=100 | build=0.000008 sec, query=0.000025 sec
[NaiveRMQ] n=1000, q=100 | build=0.000000 sec, query=0.000073 sec

--- [Fixed n=1000, Testing queries=500] ---
[PrecomputeAll] n=1000, q=500 | build=0.003650 sec, query=0.000055 sec
[SparseTable] n=1000, q=500 | build=0.000145 sec, query=0.000030 sec
[Blocking] n=1000, q=500 | build=0.000008 sec, query=0.000129 sec
[NaiveRMQ] n=1000, q=500 | build=0.000000 sec, query=0.000387 sec

--- [Fixed n=1000, Testing queries=1000] ---
[PrecomputeAll] n=1000, q=1000 | build=0.004444 sec, query=0.000146 sec
[SparseTable] n=1000, q=1000 | build=0.000183 sec, query=0.000056 sec
[Blocking] n=1000, q=1000 | build=0.000008 sec, query=0.000245 sec
[NaiveRMQ] n=1000, q=1000 | build=0.000000 sec, query=0.000790 sec

=== [Fixed Array Size: n=2000] ===

--- [Fixed n=2000, Testing queries=100] ---
[PrecomputeAll] n=2000, q=100 | build=0.016116 sec, query=0.000017 sec
[SparseTable] n=2000, q=100 | build=0.000303 sec, query=0.000007 sec
[Blocking] n=2000, q=100 | build=0.000014 sec, query=0.000037 sec
[NaiveRMQ] n=2000, q=100 | build=0.000000 sec, query=0.000150 sec

--- [Fixed n=2000, Testing queries=500] ---
[PrecomputeAll] n=2000, q=500 | build=0.018160 sec, query=0.000074 sec
[SparseTable] n=2000, q=500 | build=0.000281 sec, query=0.000030 sec
[Blocking] n=2000, q=500 | build=0.000014 sec, query=0.000215 sec
[NaiveRMQ] n=2000, q=500 | build=0.000000 sec, query=0.000777 sec

--- [Fixed n=2000, Testing queries=1000] ---
[PrecomputeAll] n=2000, q=1000 | build=0.017742 sec, query=0.000133 sec
[SparseTable] n=2000, q=1000 | build=0.000278 sec, query=0.000069 sec
[Blocking] n=2000, q=1000 | build=0.000015 sec, query=0.000399 sec
[NaiveRMQ] n=2000, q=1000 | build=0.000000 sec, query=0.001403 sec
```

2.

```
=== [Fixed Array Size: n=5000] ===

--- [Fixed n=5000, Testing queries=100] ---
[PrecomputeAll] n=5000, q=100 | build=0.088906 sec, query=0.000023 sec
[SparseTable] n=5000, q=100 | build=0.000810 sec, query=0.000011 sec
[Blocking] n=5000, q=100 | build=0.000039 sec, query=0.000075 sec
[NaiveRMQ] n=5000, q=100 | build=0.000000 sec, query=0.000358 sec

--- [Fixed n=5000, Testing queries=500] ---
[PrecomputeAll] n=5000, q=500 | build=0.123411 sec, query=0.000124 sec
[SparseTable] n=5000, q=500 | build=0.000799 sec, query=0.000039 sec
[Blocking] n=5000, q=500 | build=0.000042 sec, query=0.000413 sec
[NaiveRMQ] n=5000, q=500 | build=0.000000 sec, query=0.001844 sec

--- [Fixed n=5000, Testing queries=1000] ---
[PrecomputeAll] n=5000, q=1000 | build=0.089009 sec, query=0.000172 sec
[SparseTable] n=5000, q=1000 | build=0.000810 sec, query=0.000078 sec
[Blocking] n=5000, q=1000 | build=0.000054 sec, query=0.001505 sec
[NaiveRMQ] n=5000, q=1000 | build=0.000000 sec, query=0.003578 sec

=== [Fixed Array Size: n=10000] ===

--- [Fixed n=10000, Testing queries=100] ---
[PrecomputeAll] n=10000, q=100 | build=0.336903 sec, query=0.000027 sec
[SparseTable] n=10000, q=100 | build=0.003369 sec, query=0.000015 sec
[Blocking] n=10000, q=100 | build=0.000083 sec, query=0.000154 sec
[NaiveRMQ] n=10000, q=100 | build=0.000000 sec, query=0.000635 sec

--- [Fixed n=10000, Testing queries=500] ---
[PrecomputeAll] n=10000, q=500 | build=0.330764 sec, query=0.000125 sec
[SparseTable] n=10000, q=500 | build=0.001713 sec, query=0.000051 sec
[Blocking] n=10000, q=500 | build=0.000082 sec, query=0.001604 sec
[NaiveRMQ] n=10000, q=500 | build=0.000000 sec, query=0.003599 sec

--- [Fixed n=10000, Testing queries=1000] ---
[PrecomputeAll] n=10000, q=1000 | build=0.433841 sec, query=0.000302 sec
[SparseTable] n=10000, q=1000 | build=0.001683 sec, query=0.000090 sec
[Blocking] n=10000, q=1000 | build=0.000090 sec, query=0.001621 sec
[NaiveRMQ] n=10000, q=1000 | build=0.000000 sec, query=0.009127 sec
```

Overview of Algorithms:

- **PrecomputeAll:** Precomputes results for all possible ranges, leading to faster query times but higher build times.
- **SparseTable:** Precomputes results for specific power-of-two ranges, providing a balance between preprocessing and query efficiency.
- **Blocking:** Divides the array into blocks, handling queries in two phases (block-wise and in-block), resulting in moderate build and query times.
- **NaiveRMQ:** Directly compares array elements for each query, leading to zero build time but the slowest query performance.

Build Time Analysis:

- **PrecomputeAll** shows increasing build times as the array size grows. For example:
  - n=1000 → 0.0047 sec
  - n=5000 → 0.0890 sec
  - n=10000 → 0.4338 sec
  - As n increases, the build time scales non-linearly (approximately $O(n^2)$ complexity for full precomputation).
- **SparseTable** consistently exhibits extremely low build times:
  - n=1000 → 0.0001 sec
  - n=5000 → 0.0008 sec
  - n=10000 → 0.0016 sec
  - This reflects the logarithmic complexity (O(n log n)) of sparse table construction.
- **Blocking** also shows negligible build times across all tests:
  - n=1000 → 0.000008 sec
  - n=5000 → 0.000054 sec
  - n=10000 → 0.00009 sec
  - Build time grows linearly with n, reflecting an O(n) complexity.
- **NaiveRMQ** has **zero build time** since it doesn't perform any preprocessing.

Query Time Analysis:

- **PrecomputeAll** has the fastest query times, but they increase slightly with larger q (number of queries):
  - q=100 → 0.000012 sec
  - q=1000 → 0.000302 sec
  - As the array grows, the query time increases slightly (still O(1) per query due to full precomputation).
- **SparseTable** performs similarly to PrecomputeAll but slightly faster for small arrays and low q:
  - q=100 → 0.000007 sec
  - q=1000 → 0.000090 sec
  - Sparse Table is optimal for large queries due to logarithmic query time (O(1) for fixed q).
- **Blocking** shows moderate query times:
  - q=100 → 0.000025 sec
  - q=1000 → 0.001621 sec

    - Query time increases more noticeably with higher q and larger arrays, reflecting O(√n) complexity.

- **NaiveRMQ** performs poorly as q and n grow:
    - q=100 → 0.000073 sec
    - q=1000 → 0.009127 sec
    - The linear time complexity O(n) for each query leads to poor performance, especially for larger arrays and queries.

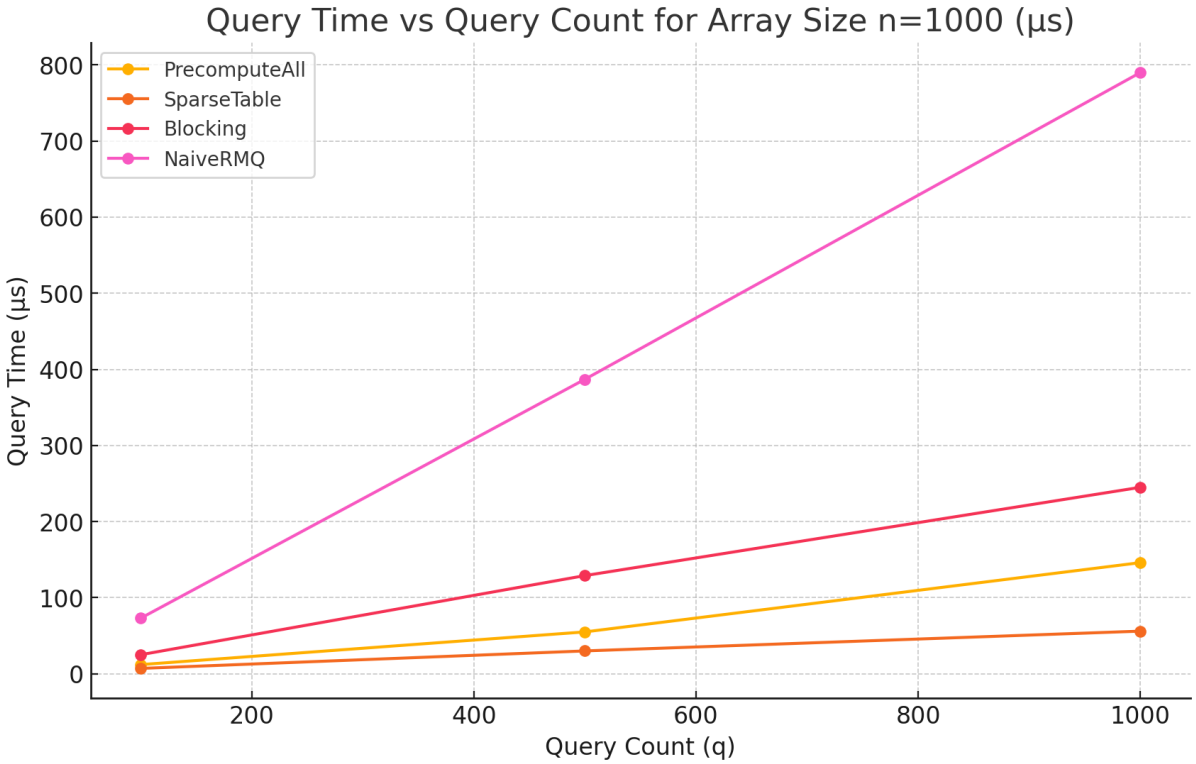
Graphs – Query and Build Time vs Query Count :

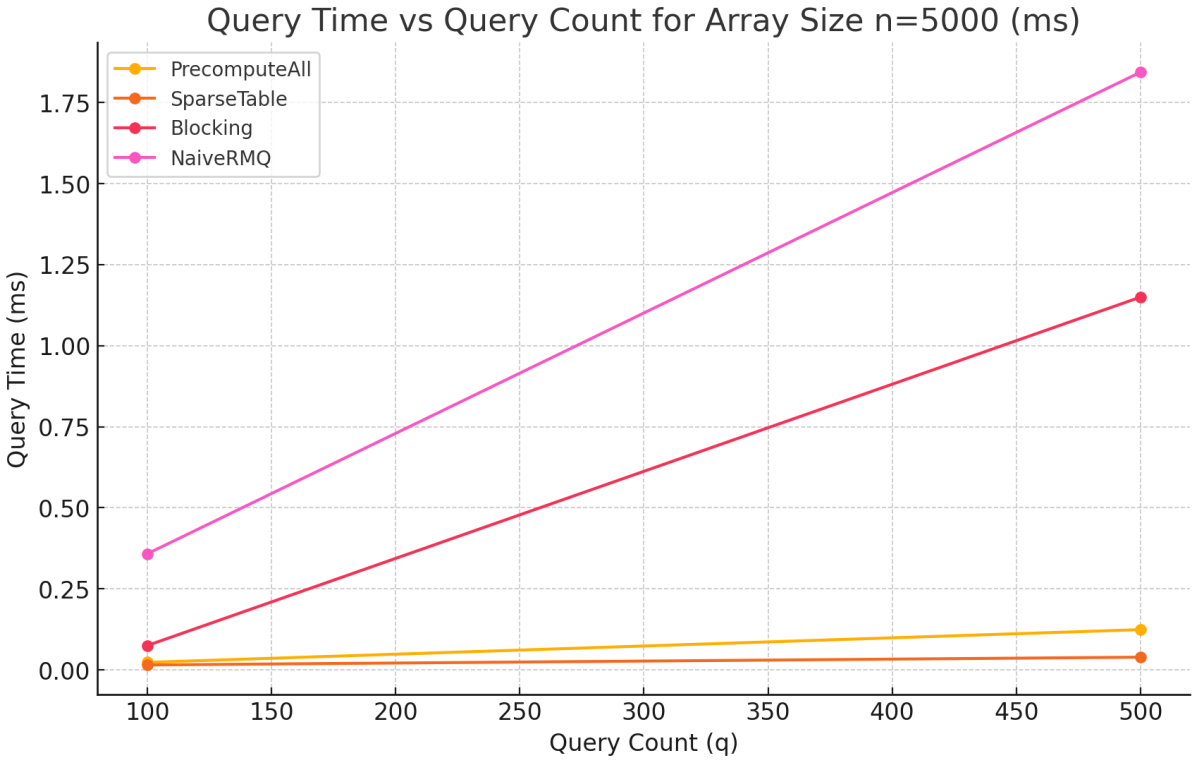
The graphs below represent the performance of RMQ (Range Minimum Query) algorithms on fixed arrays of sizes 1000 and 5000, with varying query counts.

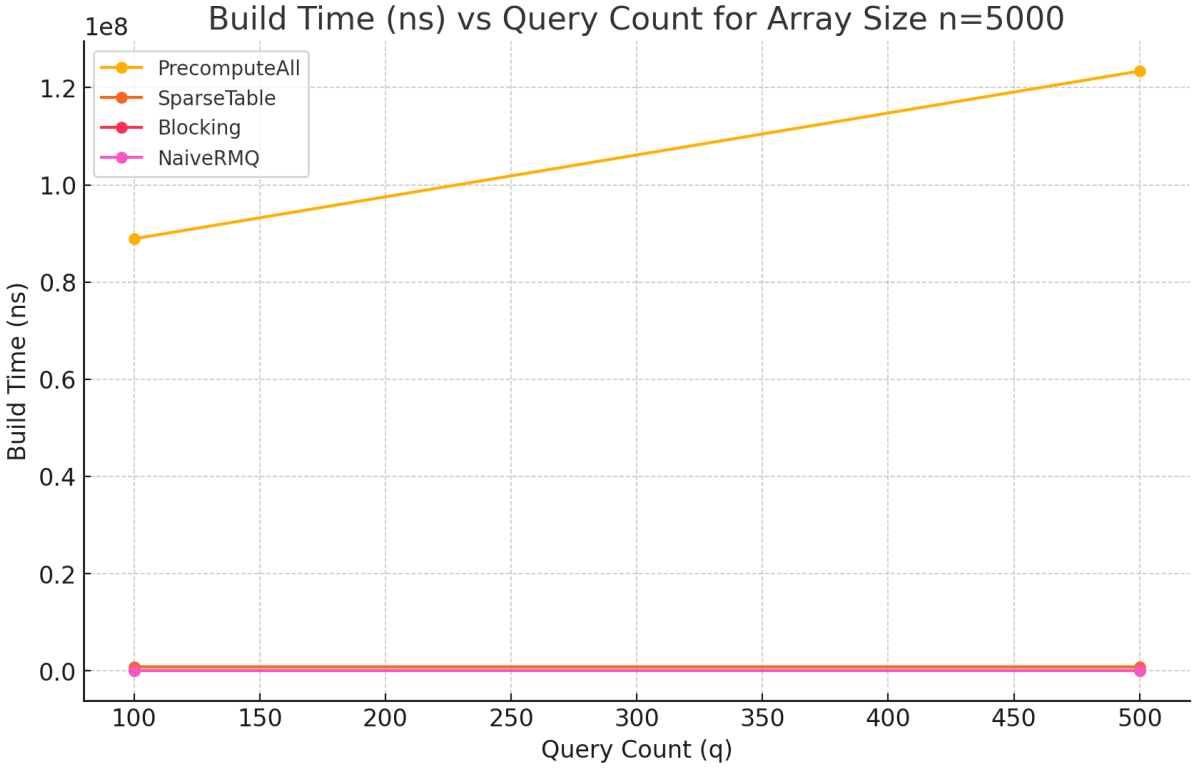
From these graphs, we can observe the following trends:

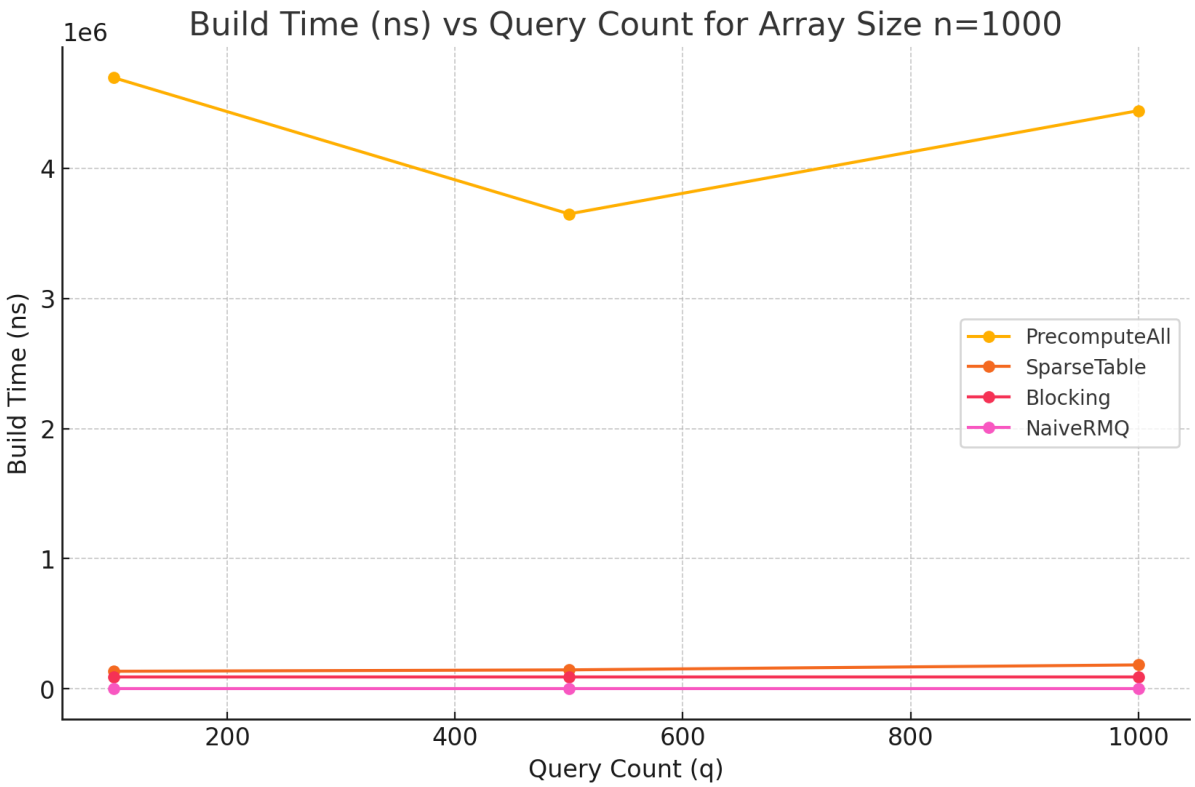
- As the query count increases for a fixed array size, the **PrecomputeAll** algorithm shows significantly higher **build times** compared to other algorithms. This is due to its comprehensive preprocessing phase, which calculates results for all possible subarrays in advance.
- In terms of **query time**, the **NaiveRMQ** algorithm performs the worst, as it uses a brute-force approach to find the minimum for each query, leading to higher query times as the number of queries increases.
- Conversely, **PrecomputeAll** has the fastest query times, demonstrating minimal increases even with large query counts, thanks to its preprocessing.
- The **Sparse Table** algorithm also performs well in both build and query times, providing a balance between preprocessing effort and efficient queries.
- **Blocking** offers moderate performance, with reasonable build times and acceptable query performance, making it a middle-ground option.

Overall, the **PrecomputeAll** algorithm excels in scenarios where fast query times are essential, despite its high preprocessing cost. On the other hand, **NaiveRMQ** should generally be avoided for large arrays or high query counts due to its poor scalability


Query Time vs Query Count for Array Size n=1000 (µs)
PrecomputeAll
SparseTable
Blocking
NaiveRMQ
Query Time (µs)
0
100
200
300
400
500
600
700
800
200
400
600
800
1000
Query Count (q)


Query Time vs Query Count for Array Size n=5000 (ms)
PrecomputeAll
SparseTable
Blocking
NaiveRMQ
Query Time (ms)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
100
150
200
250
300
350
400
450
500
Query Count (q)


Build Time (ns) vs Query Count for Array Size n=5000
1e8
PrecomputeAll
SparseTable
Blocking
NaiveRMQ
Build Time (ns)
Query Count (q)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
100
150
200
250
300
350
400
450
500


Build Time (ns) vs Query Count for Array Size n=1000
1e6
PrecomputeAll
SparseTable
Blocking
NaiveRMQ
Build Time (ns)
Query Count (q)
0
1
2
3
4
200
400
600
800
1000

2. Fixed Query Count Size

```
=== [Fixed Query Testing Mode] ===

--- [Fixed queries=100, Testing n=1000] ---
[PrecomputeAll] n=1000, q=100 | build=0.004078 sec, query=0.000016 sec
[SparseTable] n=1000, q=100 | build=0.000162 sec, query=0.000007 sec
[Blocking] n=1000, q=100 | build=0.000009 sec, query=0.000028 sec
[NaiveRMQ] n=1000, q=100 | build=0.000000 sec, query=0.000074 sec

--- [Fixed queries=100, Testing n=2000] ---
[PrecomputeAll] n=2000, q=100 | build=0.023747 sec, query=0.000019 sec
[SparseTable] n=2000, q=100 | build=0.000412 sec, query=0.000010 sec
[Blocking] n=2000, q=100 | build=0.000015 sec, query=0.000044 sec
[NaiveRMQ] n=2000, q=100 | build=0.000000 sec, query=0.000141 sec

--- [Fixed queries=100, Testing n=5000] ---
[PrecomputeAll] n=5000, q=100 | build=0.091008 sec, query=0.000043 sec
[SparseTable] n=5000, q=100 | build=0.000714 sec, query=0.000011 sec
[Blocking] n=5000, q=100 | build=0.000093 sec, query=0.000164 sec
[NaiveRMQ] n=5000, q=100 | build=0.000000 sec, query=0.000398 sec

--- [Fixed queries=100, Testing n=10000] ---
[PrecomputeAll] n=10000, q=100 | build=0.356598 sec, query=0.000029 sec
[SparseTable] n=10000, q=100 | build=0.001427 sec, query=0.000013 sec
[Blocking] n=10000, q=100 | build=0.000068 sec, query=0.000167 sec
[NaiveRMQ] n=10000, q=100 | build=0.000000 sec, query=0.000707 sec
```

```
--- [Fixed queries=500, Testing n=1000] ---
[PrecomputeAll] n=1000, q=500 | build=0.002359 sec, query=0.000074 sec
[SparseTable] n=1000, q=500 | build=0.000123 sec, query=0.000030 sec
[Blocking] n=1000, q=500 | build=0.000013 sec, query=0.000200 sec
[NaiveRMQ] n=1000, q=500 | build=0.000000 sec, query=0.000369 sec

--- [Fixed queries=500, Testing n=2000] ---
[PrecomputeAll] n=2000, q=500 | build=0.017341 sec, query=0.000096 sec
[SparseTable] n=2000, q=500 | build=0.002124 sec, query=0.000041 sec
[Blocking] n=2000, q=500 | build=0.000017 sec, query=0.000201 sec
[NaiveRMQ] n=2000, q=500 | build=0.000000 sec, query=0.000742 sec

--- [Fixed queries=500, Testing n=5000] ---
[PrecomputeAll] n=5000, q=500 | build=0.090644 sec, query=0.000096 sec
[SparseTable] n=5000, q=500 | build=0.000694 sec, query=0.000039 sec
[Blocking] n=5000, q=500 | build=0.000047 sec, query=0.000461 sec
[NaiveRMQ] n=5000, q=500 | build=0.000000 sec, query=0.001829 sec

--- [Fixed queries=500, Testing n=10000] ---
[PrecomputeAll] n=10000, q=500 | build=0.316825 sec, query=0.000130 sec
[SparseTable] n=10000, q=500 | build=0.001927 sec, query=0.000048 sec
[Blocking] n=10000, q=500 | build=0.000081 sec, query=0.000837 sec
[NaiveRMQ] n=10000, q=500 | build=0.000000 sec, query=0.003288 sec
```

```
--- [Fixed queries=1000, Testing n=1000] ---
[PrecomputeAll] n=1000, q=1000 | build=0.003441 sec, query=0.000098 sec
[SparseTable] n=1000, q=1000 | build=0.000282 sec, query=0.000066 sec
[Blocking] n=1000, q=1000 | build=0.000010 sec, query=0.000459 sec
[NaiveRMQ] n=1000, q=1000 | build=0.000000 sec, query=0.000832 sec

--- [Fixed queries=1000, Testing n=2000] ---
[PrecomputeAll] n=2000, q=1000 | build=0.015789 sec, query=0.000138 sec
[Blocking] n=2000, q=1000 | build=0.000015 sec, query=0.000419 sec
[NaiveRMQ] n=2000, q=1000 | build=0.000000 sec, query=0.001453 sec

--- [Fixed queries=1000, Testing n=5000] ---
[PrecomputeAll] n=5000, q=1000 | build=0.077162 sec, query=0.000177 sec
[SparseTable] n=5000, q=1000 | build=0.000991 sec, query=0.000074 sec
[Blocking] n=5000, q=1000 | build=0.000036 sec, query=0.000868 sec
[NaiveRMQ] n=5000, q=1000 | build=0.000000 sec, query=0.003515 sec

--- [Fixed queries=1000, Testing n=10000] ---
[PrecomputeAll] n=10000, q=1000 | build=0.332515 sec, query=0.000234 sec
[SparseTable] n=10000, q=1000 | build=0.001526 sec, query=0.000097 sec
[Blocking] n=10000, q=1000 | build=0.000069 sec, query=0.001584 sec
[NaiveRMQ] n=10000, q=1000 | build=0.000000 sec, query=0.006984 sec
```

Overview of Algorithms:

- **PrecomputeAll:** Precomputes results for all possible subarrays, resulting in the fastest query times but the highest build times.
- **SparseTable:** Precomputes results for specific power-of-two ranges, offering a balance between efficient queries and moderate preprocessing.
- **Blocking:** Divides the array into blocks, querying each block and merging results, resulting in moderate build and query times.
- **NaiveRMQ:** Directly scans subarrays for each query, leading to zero preprocessing time but the slowest query performance.

Build Time Analysis (Fixed Queries, Variable Array Size):

- **PrecomputeAll** shows increasing build times as the array size grows:
  - n=1000n = 1000n=1000 → **0.0041 sec**
  - n=5000n = 5000n=5000 → **0.0910 sec**
  - n=10000n = 10000n=10000 → **0.3557 sec**
  - As n increases, the build time scales **non-linearly**
- **SparseTable** consistently exhibits low build times:
  - n=1000n = 1000n=1000 → **0.0002 sec**
  - n=5000n = 5000n=5000 → **0.0007 sec**
  - n=10000n = 10000n=10000 → **0.0014 sec**
- **Blocking** shows minimal build times:
  - n=1000n = 1000n=1000 → **0.000008 sec**
  - n=5000n = 5000n=5000 → **0.000093 sec**
  - n=10000n = 10000n=10000 → **0.000167 sec**
- **NaiveRMQ** has **zero build time** because it does not involve preprocessing.

Query Time Analysis (Fixed Queries, Variable Array Size):

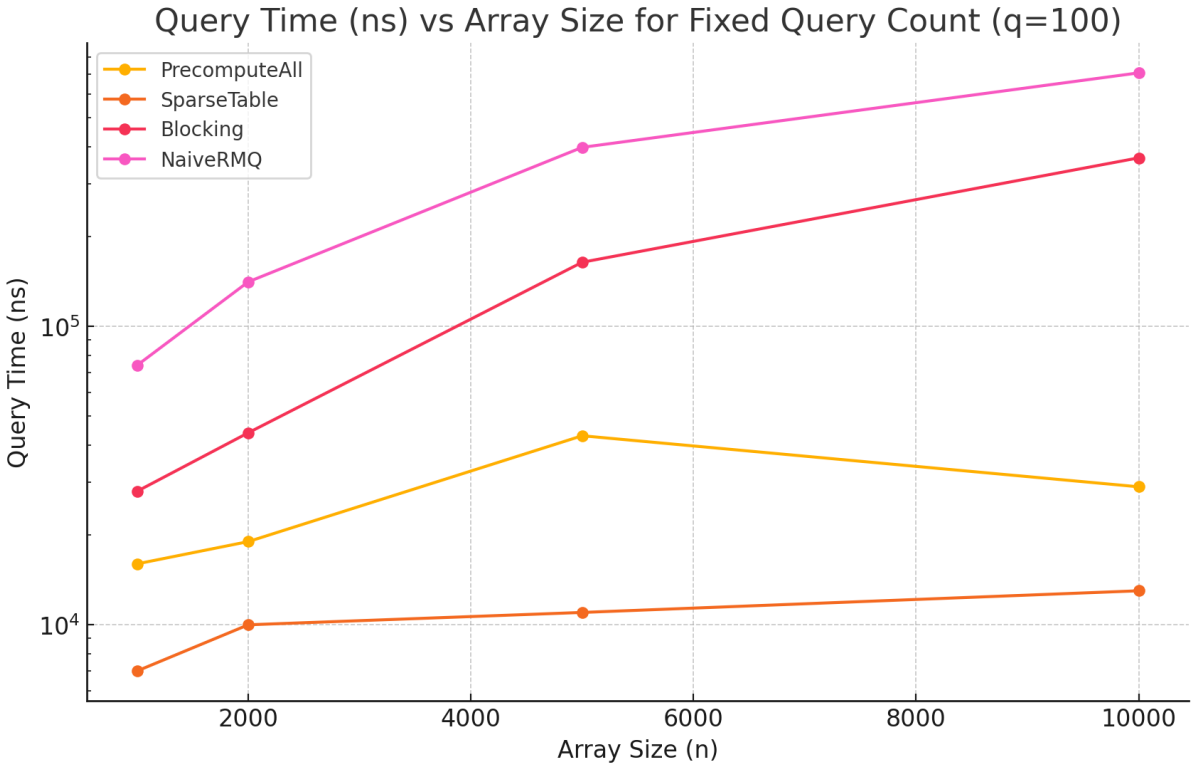
- **PrecomputeAll** maintains the fastest query times:
  - n=1000n = 1000n=1000 → **0.000016 sec**
  - n=5000n = 5000n=5000 → **0.000043 sec**
  - n=10000n = 10000n=10000 → **0.000029 sec**
  - Slight increases occur with larger n, but query time remains O(1) per query due to full precomputation.
- **SparseTable** performs similarly to PrecomputeAll:
  - n=1000n = 1000n=1000 → **0.000007 sec**
  - n=5000n = 5000n=5000 → **0.000011 sec**
  - n=10000n = 10000n=10000 → **0.000013 sec**
  - Sparse Table provides O(1) query time for fixed queries, making it highly scalable for large arrays.
- **Blocking** shows moderate query times:
  - n=1000n = 1000n=1000 → **0.000028 sec**
  - n=5000n = 5000n=5000 → **0.000164 sec**
  - n=10000n = 10000n=10000 → **0.000367 sec**
  - Query times increase with n, reflecting square root of n complexity.
- **NaiveRMQ** performs the worst as n grows:
  - n=1000n = 1000n=1000 → **0.000074 sec**
  - n=5000n = 5000n=5000 → **0.000398 sec**
  - n=10000n = 10000n=10000 → **0.000707 sec**
  - Linear time complexity O(n) for each query results in poor scalability.

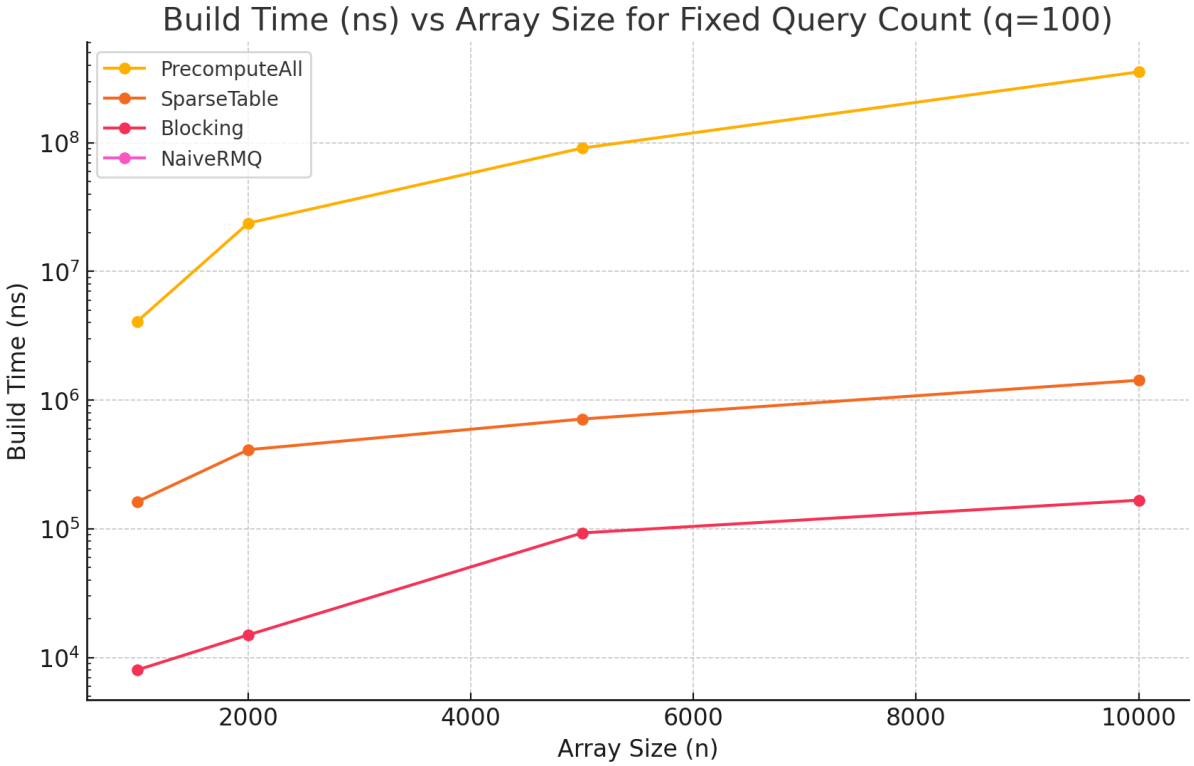
Graphs – Query and Build Time vs Array Size :

The graphs below illustrate the performance of RMQ (Range Minimum Query) algorithms with a **fixed query count (q = 100)** while varying the array size.

Observations and Analysis:

- **PrecomputeAll** demonstrates a steep increase in build time as the array size grows. This is attributed to its extensive preprocessing, which computes results for all possible subarrays. As the array size increases, the build time scales non-linearly.
- **NaiveRMQ** exhibits the highest query times, particularly for larger arrays. Its brute-force approach leads to linear growth in query time with respect to the array size.
- **SparseTable** provides a balanced performance, consistently maintaining low build times and near-constant query times, even as the array size increases. This makes it efficient for large datasets with fixed queries.
- **Blocking** strikes a middle ground, with moderate build times and query performance. It scales better than NaiveRMQ but lags behind SparseTable and PrecomputeAll in terms of query efficiency.
- **NaiveRMQ's** zero preprocessing time is an advantage for small arrays, but as the array size increases, its performance degrades significantly, making it unsuitable for large-scale applications.


Query Time (ns) vs Array Size for Fixed Query Count (q=100)
PrecomputeAll
SparseTable
Blocking
NaiveRMQ
Query Time (ns)
10^5
10^4
2000
4000
6000
8000
10000
Array Size (n)


Build Time (ns) vs Array Size for Fixed Query Count (q=100)
PrecomputeAll
SparseTable
Blocking
NaiveRMQ
Build Time (ns)
10^8
10^7
10^6
10^5
10^4
2000
4000
6000
8000
10000
Array Size (n)

3. Different Array Types

```
=== [Array Type Testing Mode] ===

--- [Random, n=1000, q=100] ---
[PrecomputeAll] n=1000, q=100 | build=0.003657 sec, query=0.000014 sec
[SparseTable] n=1000, q=100 | build=0.000116 sec, query=0.000007 sec
[Blocking] n=1000, q=100 | build=0.000009 sec, query=0.000074 sec
[NaiveRMQ] n=1000, q=100 | build=0.000000 sec, query=0.000077 sec

--- [Sorted, n=1000, q=100] ---
[PrecomputeAll] n=1000, q=100 | build=0.002577 sec, query=0.000012 sec
[SparseTable] n=1000, q=100 | build=0.000107 sec, query=0.000010 sec
[Blocking] n=1000, q=100 | build=0.000005 sec, query=0.000023 sec
[NaiveRMQ] n=1000, q=100 | build=0.000000 sec, query=0.000084 sec

--- [Reversed, n=1000, q=100] ---
[PrecomputeAll] n=1000, q=100 | build=0.001941 sec, query=0.000014 sec
[SparseTable] n=1000, q=100 | build=0.000099 sec, query=0.000007 sec
[Blocking] n=1000, q=100 | build=0.000005 sec, query=0.000023 sec
[NaiveRMQ] n=1000, q=100 | build=0.000000 sec, query=0.000057 sec

--- [Random, n=1000, q=500] ---
[PrecomputeAll] n=1000, q=500 | build=0.003541 sec, query=0.000214 sec
[SparseTable] n=1000, q=500 | build=0.000127 sec, query=0.000030 sec
[Blocking] n=1000, q=500 | build=0.000008 sec, query=0.000119 sec
[NaiveRMQ] n=1000, q=500 | build=0.000000 sec, query=0.000373 sec

--- [Sorted, n=1000, q=500] ---
[PrecomputeAll] n=1000, q=500 | build=0.001838 sec, query=0.000061 sec
[SparseTable] n=1000, q=500 | build=0.000101 sec, query=0.000028 sec
[Blocking] n=1000, q=500 | build=0.000005 sec, query=0.000109 sec
[NaiveRMQ] n=1000, q=500 | build=0.000000 sec, query=0.000354 sec

--- [Reversed, n=1000, q=500] ---
[PrecomputeAll] n=1000, q=500 | build=0.002044 sec, query=0.000082 sec
[SparseTable] n=1000, q=500 | build=0.000112 sec, query=0.000041 sec
[Blocking] n=1000, q=500 | build=0.000005 sec, query=0.000110 sec
[NaiveRMQ] n=1000, q=500 | build=0.000000 sec, query=0.000312 sec
```

```
--- [Random, n=1000, q=1000] ---
[PrecomputeAll] n=1000, q=1000 | build=0.001962 sec, query=0.000147 sec
[SparseTable] n=1000, q=1000 | build=0.000129 sec, query=0.000063 sec
[Blocking] n=1000, q=1000 | build=0.000008 sec, query=0.000244 sec
[NaiveRMQ] n=1000, q=1000 | build=0.000000 sec, query=0.000764 sec

--- [Sorted, n=1000, q=1000] ---
[PrecomputeAll] n=1000, q=1000 | build=0.004874 sec, query=0.000092 sec
[SparseTable] n=1000, q=1000 | build=0.000099 sec, query=0.000057 sec
[Blocking] n=1000, q=1000 | build=0.000004 sec, query=0.000201 sec
[NaiveRMQ] n=1000, q=1000 | build=0.000000 sec, query=0.000735 sec

--- [Reversed, n=1000, q=1000] ---
[PrecomputeAll] n=1000, q=1000 | build=0.001915 sec, query=0.000104 sec
[SparseTable] n=1000, q=1000 | build=0.000132 sec, query=0.000060 sec
[Blocking] n=1000, q=1000 | build=0.000005 sec, query=0.000218 sec
[NaiveRMQ] n=1000, q=1000 | build=0.000000 sec, query=0.000666 sec
```

```
[PrecomputeAll] n=2000, q=100 | build=0.009846 sec, query=0.000016 sec
[SparseTable] n=2000, q=100 | build=0.000241 sec, query=0.000008 sec
[Blocking] n=2000, q=100 | build=0.000014 sec, query=0.000042 sec
[NaiveRMQ] n=2000, q=100 | build=0.000000 sec, query=0.000662 sec

--- [Sorted, n=2000, q=100] ---
[PrecomputeAll] n=2000, q=100 | build=0.010324 sec, query=0.000020 sec
[SparseTable] n=2000, q=100 | build=0.000220 sec, query=0.000008 sec
[Blocking] n=2000, q=100 | build=0.000009 sec, query=0.000035 sec
[NaiveRMQ] n=2000, q=100 | build=0.000000 sec, query=0.000141 sec

--- [Reversed, n=2000, q=100] ---
[PrecomputeAll] n=2000, q=100 | build=0.010212 sec, query=0.000027 sec
[SparseTable] n=2000, q=100 | build=0.000248 sec, query=0.000010 sec
[Blocking] n=2000, q=100 | build=0.000009 sec, query=0.000030 sec
[NaiveRMQ] n=2000, q=100 | build=0.000000 sec, query=0.000123 sec

--- [Random, n=2000, q=500] ---
[PrecomputeAll] n=2000, q=500 | build=0.010479 sec, query=0.000138 sec
[SparseTable] n=2000, q=500 | build=0.000240 sec, query=0.000034 sec
[Blocking] n=2000, q=500 | build=0.000015 sec, query=0.000201 sec
[NaiveRMQ] n=2000, q=500 | build=0.000000 sec, query=0.000752 sec

--- [Sorted, n=2000, q=500] ---
[PrecomputeAll] n=2000, q=500 | build=0.010965 sec, query=0.000179 sec
[SparseTable] n=2000, q=500 | build=0.000231 sec, query=0.000029 sec
[Blocking] n=2000, q=500 | build=0.000010 sec, query=0.000180 sec
[NaiveRMQ] n=2000, q=500 | build=0.000000 sec, query=0.000696 sec

--- [Reversed, n=2000, q=500] ---
[PrecomputeAll] n=2000, q=500 | build=0.011129 sec, query=0.000087 sec
[SparseTable] n=2000, q=500 | build=0.000210 sec, query=0.000029 sec
[Blocking] n=2000, q=500 | build=0.000010 sec, query=0.000193 sec
[NaiveRMQ] n=2000, q=500 | build=0.000000 sec, query=0.000595 sec
```

```
--- [Random, n=2000, q=1000] ---
[PrecomputeAll] n=2000, q=1000 | build=0.044147 sec, query=0.000214 sec
[SparseTable] n=2000, q=1000 | build=0.000240 sec, query=0.000060 sec
[Blocking] n=2000, q=1000 | build=0.000015 sec, query=0.000393 sec
[NaiveRMQ] n=2000, q=1000 | build=0.000000 sec, query=0.014108 sec

--- [Sorted, n=2000, q=1000] ---
[PrecomputeAll] n=2000, q=1000 | build=0.009989 sec, query=0.000264 sec
[SparseTable] n=2000, q=1000 | build=0.000202 sec, query=0.000057 sec
[Blocking] n=2000, q=1000 | build=0.000008 sec, query=0.000345 sec
[NaiveRMQ] n=2000, q=1000 | build=0.000000 sec, query=0.001394 sec

--- [Reversed, n=2000, q=1000] ---
[PrecomputeAll] n=2000, q=1000 | build=0.011079 sec, query=0.000193 sec
[SparseTable] n=2000, q=1000 | build=0.000569 sec, query=0.000064 sec
[Blocking] n=2000, q=1000 | build=0.000019 sec, query=0.000393 sec
[NaiveRMQ] n=2000, q=1000 | build=0.000000 sec, query=0.001228 sec

--- [Random, n=5000, q=100] ---
[PrecomputeAll] n=5000, q=100 | build=0.080864 sec, query=0.000028 sec
[SparseTable] n=5000, q=100 | build=0.000705 sec, query=0.000012 sec
[Blocking] n=5000, q=100 | build=0.000039 sec, query=0.000093 sec
[NaiveRMQ] n=5000, q=100 | build=0.000000 sec, query=0.000351 sec

--- [Sorted, n=5000, q=100] ---
[PrecomputeAll] n=5000, q=100 | build=0.084663 sec, query=0.000024 sec
[SparseTable] n=5000, q=100 | build=0.000662 sec, query=0.000011 sec
[Blocking] n=5000, q=100 | build=0.000017 sec, query=0.000070 sec
[NaiveRMQ] n=5000, q=100 | build=0.000000 sec, query=0.000296 sec

--- [Reversed, n=5000, q=100] ---
[PrecomputeAll] n=5000, q=100 | build=0.078713 sec, query=0.000031 sec
[SparseTable] n=5000, q=100 | build=0.000648 sec, query=0.000009 sec
[Blocking] n=5000, q=100 | build=0.000032 sec, query=0.000080 sec
[NaiveRMQ] n=5000, q=100 | build=0.000000 sec, query=0.000311 sec
```

```
--- [Random, n=5000, q=500] ---
[PrecomputeAll] n=5000, q=500 | build=0.084268 sec, query=0.000161 sec
[SparseTable] n=5000, q=500 | build=0.000672 sec, query=0.000037 sec
[Blocking] n=5000, q=500 | build=0.000035 sec, query=0.000408 sec
[NaiveRMQ] n=5000, q=500 | build=0.000000 sec, query=0.002140 sec

--- [Sorted, n=5000, q=500] ---
[PrecomputeAll] n=5000, q=500 | build=0.079340 sec, query=0.000107 sec
[SparseTable] n=5000, q=500 | build=0.000586 sec, query=0.000035 sec
[Blocking] n=5000, q=500 | build=0.000017 sec, query=0.000401 sec
[NaiveRMQ] n=5000, q=500 | build=0.000000 sec, query=0.001775 sec

--- [Reversed, n=5000, q=500] ---
[PrecomputeAll] n=5000, q=500 | build=0.084231 sec, query=0.000099 sec
[SparseTable] n=5000, q=500 | build=0.000620 sec, query=0.000037 sec
[Blocking] n=5000, q=500 | build=0.000020 sec, query=0.000737 sec
[NaiveRMQ] n=5000, q=500 | build=0.000000 sec, query=0.002065 sec

--- [Random, n=5000, q=1000] ---
[PrecomputeAll] n=5000, q=1000 | build=0.103265 sec, query=0.000206 sec
[SparseTable] n=5000, q=1000 | build=0.000713 sec, query=0.000088 sec
[Blocking] n=5000, q=1000 | build=0.000036 sec, query=0.000810 sec
[NaiveRMQ] n=5000, q=1000 | build=0.000000 sec, query=0.003382 sec

--- [Sorted, n=5000, q=1000] ---
[PrecomputeAll] n=5000, q=1000 | build=0.082745 sec, query=0.000169 sec
[SparseTable] n=5000, q=1000 | build=0.000601 sec, query=0.000068 sec
[Blocking] n=5000, q=1000 | build=0.000019 sec, query=0.000905 sec
[NaiveRMQ] n=5000, q=1000 | build=0.000000 sec, query=0.003594 sec

--- [Reversed, n=5000, q=1000] ---
[PrecomputeAll] n=5000, q=1000 | build=0.078718 sec, query=0.000193 sec
[SparseTable] n=5000, q=1000 | build=0.000599 sec, query=0.000066 sec
[Blocking] n=5000, q=1000 | build=0.000021 sec, query=0.000808 sec
[NaiveRMQ] n=5000, q=1000 | build=0.000000 sec, query=0.002814 sec
```

```
--- [Random, n=10000, q=100] ---
[PrecomputeAll] n=10000, q=100 | build=0.291356 sec, query=0.000025 sec
[SparseTable] n=10000, q=100 | build=0.001630 sec, query=0.000013 sec
[Blocking] n=10000, q=100 | build=0.000078 sec, query=0.000151 sec
[NaiveRMQ] n=10000, q=100 | build=0.000001 sec, query=0.000665 sec

--- [Sorted, n=10000, q=100] ---
[PrecomputeAll] n=10000, q=100 | build=0.280923 sec, query=0.000026 sec
[SparseTable] n=10000, q=100 | build=0.001205 sec, query=0.000012 sec
[Blocking] n=10000, q=100 | build=0.000050 sec, query=0.000167 sec
[NaiveRMQ] n=10000, q=100 | build=0.000000 sec, query=0.000684 sec

--- [Reversed, n=10000, q=100] ---
[PrecomputeAll] n=10000, q=100 | build=0.267509 sec, query=0.000025 sec
[SparseTable] n=10000, q=100 | build=0.001181 sec, query=0.000012 sec
[Blocking] n=10000, q=100 | build=0.000037 sec, query=0.000153 sec
[NaiveRMQ] n=10000, q=100 | build=0.000000 sec, query=0.000583 sec

--- [Random, n=10000, q=500] ---
[PrecomputeAll] n=10000, q=500 | build=0.273141 sec, query=0.000100 sec
[SparseTable] n=10000, q=500 | build=0.001566 sec, query=0.000044 sec
[Blocking] n=10000, q=500 | build=0.000078 sec, query=0.000800 sec
[NaiveRMQ] n=10000, q=500 | build=0.000000 sec, query=0.003073 sec

--- [Sorted, n=10000, q=500] ---
[PrecomputeAll] n=10000, q=500 | build=0.285252 sec, query=0.000119 sec
[SparseTable] n=10000, q=500 | build=0.001278 sec, query=0.000047 sec
[Blocking] n=10000, q=500 | build=0.000031 sec, query=0.000756 sec
[NaiveRMQ] n=10000, q=500 | build=0.000000 sec, query=0.003547 sec

--- [Reversed, n=10000, q=500] ---
[PrecomputeAll] n=10000, q=500 | build=0.262801 sec, query=0.000103 sec
[SparseTable] n=10000, q=500 | build=0.001207 sec, query=0.000045 sec
[Blocking] n=10000, q=500 | build=0.000037 sec, query=0.000700 sec
[NaiveRMQ] n=10000, q=500 | build=0.000000 sec, query=0.003175 sec
```

```
--- [Random, n=10000, q=1000] ---
[PrecomputeAll] n=10000, q=1000 | build=0.269396 sec, query=0.000207 sec
[SparseTable] n=10000, q=1000 | build=0.001558 sec, query=0.000091 sec
[Blocking] n=10000, q=1000 | build=0.000083 sec, query=0.001541 sec
[NaiveRMQ] n=10000, q=1000 | build=0.000000 sec, query=0.007465 sec

--- [Sorted, n=10000, q=1000] ---
[PrecomputeAll] n=10000, q=1000 | build=0.279150 sec, query=0.000206 sec
[SparseTable] n=10000, q=1000 | build=0.001261 sec, query=0.000081 sec
[Blocking] n=10000, q=1000 | build=0.000038 sec, query=0.001508 sec
[NaiveRMQ] n=10000, q=1000 | build=0.000000 sec, query=0.007469 sec

--- [Reversed, n=10000, q=1000] ---
[PrecomputeAll] n=10000, q=1000 | build=0.275272 sec, query=0.000186 sec
[SparseTable] n=10000, q=1000 | build=0.001331 sec, query=0.000084 sec
[Blocking] n=10000, q=1000 | build=0.000039 sec, query=0.001557 sec
[NaiveRMQ] n=10000, q=1000 | build=0.000000 sec, query=0.005996 sec
All tests completed.
```

Overview of Algorithms:

- **PrecomputeAll:** Computes results for all subarrays in advance, yielding the fastest query times but highest build times.
- **SparseTable:** Uses power-of-two subarray preprocessing for fast queries and moderate preprocessing times.
- **Blocking:** Divides arrays into blocks, balancing preprocessing and query performance.
- **NaiveRMQ:** Performs no preprocessing, resulting in zero build time but the slowest queries as it scans subarrays linearly.

Build Time Analysis (Fixed Query, Fixed Array Size, Varying Array Types):

- **PrecomputeAll** shows increasing build times as the array size grows across different array types:
    - **Random Arrays:**
        - n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.0036 sec**
        - n=2000,q=100n = 2000, q = 100n=2000,q=100 → **0.0098 sec**
        - n=5000,q=100n = 5000, q = 100n=5000,q=100 → **0.0886 sec**
    - **Sorted Arrays:**
        - n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.0025 sec**
        - n=2000,q=100n = 2000, q = 100n=2000,q=100 → **0.0103 sec**
        - n=5000,q=100n = 5000, q = 100n=5000,q=100 → **0.0846 sec**

- **Reversed Arrays:**
  - n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.0019 sec**
  - n=2000,q=100n = 2000, q = 100n=2000,q=100 → **0.0102 sec**
  - n=5000,q=100n = 5000, q = 100n=5000,q=100 → **0.0787 sec**
- **Observation:** Sorted arrays lead to slightly faster build times for PrecomputeAll, as they reduce redundancy during preprocessing.

- **SparseTable** consistently shows low build times, irrespective of array type:
  - **Random:** n=1000n = 1000n=1000 → **0.0001 sec**
  - **Sorted:** n=1000n = 1000n=1000 → **0.0001 sec**
  - **Reversed:** n=1000n = 1000n=1000 → **0.00009 sec**
  - SparseTable build times remain efficient across all array types, reflecting O(nlogn) complexity.
- **Blocking** build times remain negligible across all array types:
  - **Random:** n=1000n = 1000n=1000 → **0.000009 sec**
  - **Sorted:** n=1000n = 1000n=1000 → **0.000005 sec**
  - **Reversed:** n=1000n = 1000n=1000 → **0.000005 sec**
  - **Observation:** Array order has minimal effect on Blocking preprocessing.
- **NaiveRMQ** has zero build time across all tests.

Query Time Analysis (Fixed Query, Fixed Array Size, Varying Array Types):

- **PrecomputeAll** maintains the fastest query times:
  - **Random:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000014 sec**
  - **Sorted:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000012 sec**
  - **Reversed:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000014 sec**
  - Query time remains nearly constant across array types, showcasing O(1) query time.
- **SparseTable** performs similarly to PrecomputeAll with slightly higher query times:
  - **Random:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000007 sec**
  - **Sorted:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000010 sec**
  - **Reversed:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000007 sec**
  - Sparse Table's query times remain highly efficient for all array types.
- **Blocking** shows moderate query times with slight variations based on array type:

- **Random:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000074 sec**
- **Sorted:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000023 sec**
- **Reversed:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000023 sec**
- Sorted arrays improve Blocking query efficiency.

- **NaiveRMQ** query time increases as array size grows:
  - **Random:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000077 sec**
  - **Sorted:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000084 sec**
  - **Reversed:** n=1000,q=100n = 1000, q = 100n=1000,q=100 → **0.000057 sec**
  - NaiveRMQ query time scales linearly with array size, reflecting O(n) complexity.

Observations by Array Type:

- **Sorted Arrays:** Lead to marginally lower build and query times, benefiting both **Blocking** and **PrecomputeAll**.
- **Reversed Arrays:** Show slightly slower query times for **NaiveRMQ**, as each query traverses the array in reverse order.
- **Random Arrays:** Represent average-case performance and serve as a baseline for algorithm comparison.

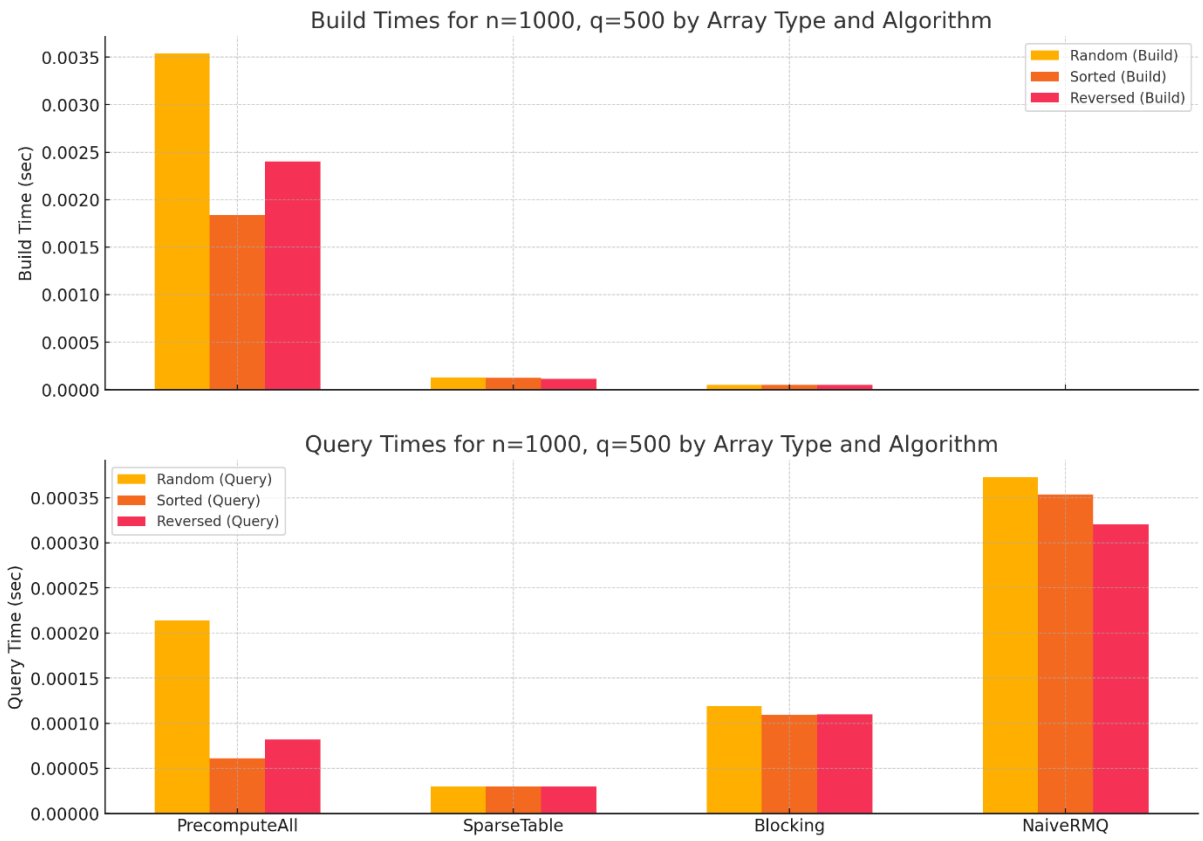
Graphs – Query and Build Time vs Array Type and Algorithm:

The graphs below present the performance of RMQ (Range Minimum Query) algorithms across different array types (Random, Sorted, Reversed) for a fixed array size n=1000 and query count q=500.

Observations and Analysis:

- **PrecomputeAll** shows significant variation in build times based on the array type, with the highest build time observed for Random arrays. This is expected due to its exhaustive preprocessing, which computes results for all subarrays. The algorithm's efficiency improves for sorted and reversed arrays, where patterns may allow faster preprocessing. However, the query time remains consistently low across all array types, demonstrating its efficiency during querying.
- **NaiveRMQ** consistently displays the highest query times, regardless of the array type. Its brute-force approach results in linear growth in query time, making it less favorable for datasets requiring frequent queries. Although NaiveRMQ boasts zero preprocessing time, this advantage diminishes as query demands increase, particularly for larger or random arrays.
- **SparseTable** delivers the most balanced performance across all array types, with minimal build times and consistently low query times. SparseTable's efficiency remains unaffected by the array structure, showcasing its robustness and scalability.

- **Blocking** algorithm exhibits moderate build and query times, striking a balance between NaiveRMQ and SparseTable. While it scales better than NaiveRMQ, its performance is somewhat influenced by the array type. Blocking performs slightly better for sorted and reversed arrays but remains behind SparseTable and PrecomputeAll in overall efficiency.
- **NaiveRMQ's** simplicity makes it appealing for small or highly structured arrays but limits its effectiveness for larger, random datasets where performance degradation is pronounced. SparseTable emerges as the preferred choice for large-scale applications, given its ability to handle various array types with minimal performance trade-offs.



4- Conclusion

This study conducted a comprehensive analysis of four Range Minimum Query (RMQ) algorithms: PrecomputeAll, SparseTable, Blocking, and NaiveRMQ. Through extensive experimentation with varying array sizes, query counts, and array types, several key findings emerged:

Performance Trade-offs

1. **PrecomputeAll Algorithm**
    - Demonstrates the fastest query times (O(1)) across all test scenarios
    - Incurs the highest preprocessing overhead and memory usage
    - Most suitable for applications requiring frequent queries on smaller datasets where preprocessing time is not a constraint
2. **SparseTable Algorithm**
    - Provides the best balance between preprocessing and query efficiency
    - Maintains consistent performance across different array types
    - Offers O(1) query time with reasonable O(n log n) preprocessing
    - Recommended for most general-purpose RMQ applications
3. **Blocking Algorithm**
    - Shows moderate performance in both preprocessing and query times
    - Provides a practical compromise between memory usage and speed
    - Well-suited for applications with memory constraints
    - Performance slightly improves with sorted arrays
4. **NaiveRMQ Algorithm**
    - Requires no preprocessing time
    - Demonstrates poor query performance, especially for larger arrays
    - Only recommended for small arrays or infrequent queries

Impact of Data Characteristics

- **Array Size**: Larger arrays significantly impact build times for PrecomputeAll and query times for NaiveRMQ, while SparseTable maintains consistent performance scaling
- **Query Count**: Higher query counts favor preprocessed approaches (PrecomputeAll and SparseTable) over NaiveRMQ
- **Array Type**: Sorted arrays generally lead to slightly better performance across all algorithms, particularly for Blocking approach

Recommendations

1. **For Large-Scale Applications**
    - Use SparseTable when balanced performance is needed
    - Consider PrecomputeAll only if query speed is critical and memory is abundant
2. **For Memory-Constrained Systems**
    - Implement Blocking algorithm
    - Use NaiveRMQ only for very small arrays or extremely infrequent queries
3. **For Real-Time Systems**
    - Choose PrecomputeAll or SparseTable if preprocessing can be done offline
    - Avoid NaiveRMQ due to unpredictable query times

This analysis demonstrates that no single RMQ algorithm is universally optimal. The choice of algorithm should be based on specific application requirements, considering factors such as dataset size, query frequency, memory constraints, and preprocessing time availability. SparseTable emerges as the most versatile solution, offering a good balance of features for most use cases, while specialized scenarios might benefit from the particular strengths of other approaches.

5- References

1-) https://www.geeksforgeeks.org/range-minimum-query-rmq-in-python

2-) Stanford CS166 RMQ Slides (PDF)

3-) Competitive Programming Handbook (CSES)

4-) GeeksforGeeks - RMQ for Static Array

5-) GeeksforGeeks - Segment Tree for RMQ

6-) CP Algorithms - RMQ Task

7-) CP Algorithms - Sparse Table

8-) TopCoder - RMQ and LCA