Range Minimum Queries (RMQ): Algorithms and Experimental Analysis

1. Problem Description

The Range Minimum Query (RMQ) problem involves efficiently finding the smallest element within a specified range of an array. It is a fundamental computational problem with significant theoretical and practical importance. At its core, the RMQ problem addresses the challenge of performing repeated queries to determine the minimum value in arbitrary subarrays of a given array.

The primary goal in RMQ is to preprocess the array, enabling rapid query responses, particularly when multiple queries are expected. This preprocessing-query trade-off makes RMQ crucial in scenarios requiring repeated minimum searches.

The problem is formally defined as follows:

Formal Definition

- **Input:** An array of elements, and a pair of indices and such that .
    - **Array(A[]):** 35, 45, 2, 18, 9, 95, 78, 55
    - **Right indices of the array(i):** 2
    - **Left index of the array(j):** 6
- **Output:** The smallest element in the subarray , i.e. ‘**2**’

Characteristics

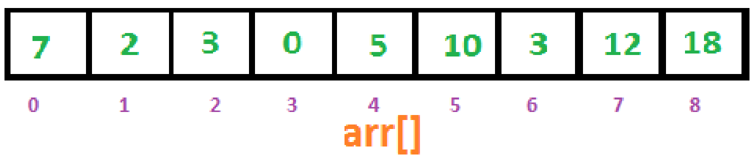
- The problem assumes that the array remains unchanged across multiple queries. This allows preprocessing of the array to optimize the query time.
- Queries are defined as , where is the array, and and are the range indices.

Applications

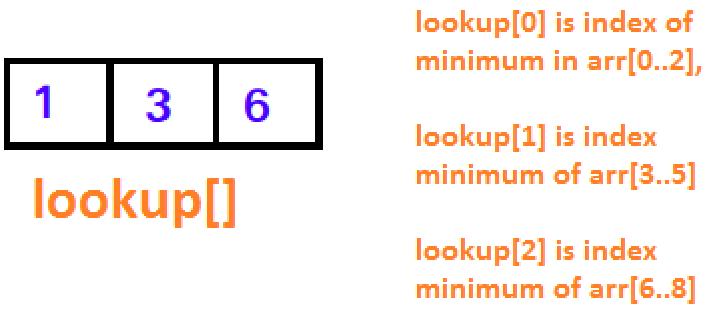
1. **Data Compression:** Efficiently finding patterns and minima in data streams.
2. **Bioinformatics:** Identifying minimum error regions in genome sequences.
3. **Geometric Algorithms:** Solving problems like lowest common ancestors in trees.
4. **Dynamic Programming:** As a subroutine in algorithms for matrix multiplication and sequence alignment.

Example

Given an array :



And the subset query of the array:


1
3
6
lookup[]
lookup[0] is index of minimum in arr[0..2],
lookup[1] is index minimum of arr[3..5]
lookup[2] is index minimum of arr[6..8]

- Query 1(Lookup 0) should return **2**, as the smallest value in the subarray **[7, 2, 3]**.
- Query 2 should return **0**, as it is the minimum in **[0, 5, 10]**.
- Query 3 should return **3**, as it is the minimum in **[3, 12, 18]**.

Problem Variants

1. **Static RMQ:** The array does not change, allowing extensive preprocessing.
2. **Dynamic RMQ:** The array allows updates between queries, requiring balanced preprocessing and query efficiency.

2. Algorithms

The RMQ problem can be approached in two main ways, each emphasizing a different optimization:

1. **Minimizing Query Time:** These approaches focus on preprocessing the array to enable extremely fast query times. They are suitable for scenarios where many queries will be performed on a largely static array. The trade-off is increased initialization time and memory usage. E.g**.** **Full preprocessing**, **Sparse table** or **hybrit Approaches**.
2. **Minimizing Initialization Time:** These methods prioritize minimal preprocessing at the cost of slower query times. They are ideal when only a few queries need to be processed, or when memory and preprocessing constraints are tight. E.g. **No Preprocessing**, **Block Partition** or **hybrit Approaches**.

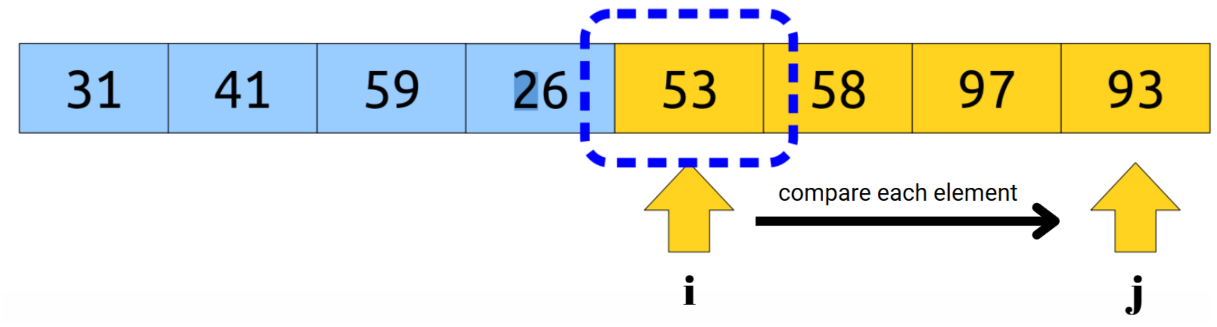
We describe the key algorithms below:

2.1 Precompute None

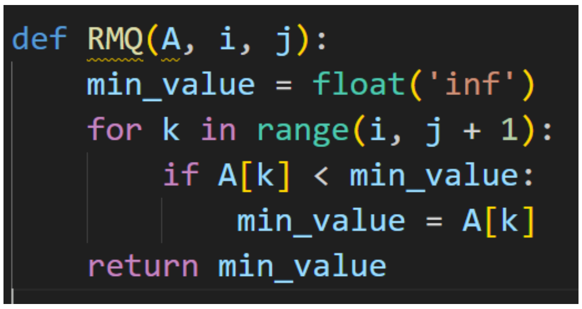
- **Description:** The "Precompute None" approach to solving the Range Minimum Query (RMQ) problem is the **simplest** and **most direct** method. It forgoes any form of preprocessing and instead performs the query operations directly on the array at runtime.

Key Steps:

1. **Query Execution:** For each query **RMQ(A,i,j)**:
   - Iterate through the elements of the array from index i to index j.
   - Compare each element to a temporary variable holding the minimum value seen so far. (**Brute Force Approach**)
   - Update the temporary variable whenever a smaller value is encountered.
2. **Return the Result**: Once the iteration completes, return the value stored in the temporary variable as the minimum of the range.



Pseudocode:



```
def RMQ(A, i, j):
    min_value = float('inf')
    for k in range(i, j + 1):
        if A[k] < min_value:
            min_value = A[k]
    return min_value
```

Theoretical Analysis:

1. **Preprocessing Time: O(1)**
   - No preprocessing is performed; the array is used as is.
2. **Query Complexity: O(n)**

- For a single query, we need to compute the minimum of $j-i+1$ elements in the range $[i,j]$. This requires $j-i+1$ comparisons in the worst case.
- If n is the size of the array and we consider the worst-case scenario, the query range could span the entire array, **i.e.**, $i=0$ and $j=n-1$. In this case, $j-i+1=n$, and the number of comparisons becomes $O(n)$.

3. **General Case for k Queries:**
   - Suppose we perform k queries. The time complexity of all queries combined will depend on the length of each range $[i_k,j_k]$. Let $l_k=j_k-i_k+1$ be the length of the k-th range.
   - The total time for all queries is:

$$T_{total}=\sum_{k=1}^{k} l_k$$

   - In the worst case, if each query spans the entire array (lk=nl_k = nlk=n), the total time complexity for **k** queries is: $\mathbf{T_{total}=k\cdot n \Rightarrow O(kn)}$
   - For large number of queries or large array sizes, this can be computationally expensive.
4. **Space Complexity: O(1)**
   - Only a few variables are used (e.g., the temporary minimum value).

Advantages:

- **Minimal Memory Usage**: Does not require additional space beyond the input array.
- **Simple Implementation**: Easy to understand and implement, with minimal code complexity.
- **No Preprocessing Overhead**: Ideal for use cases where preprocessing is not feasible due to time or system constraints.

Disadvantages:

- **Slow Query Times**: Each query requires a linear scan, making this approach inefficient for large arrays or frequent queries.
- **Lack of Scalability:** As the array size grows or the number of queries increases, the time cost becomes significant.

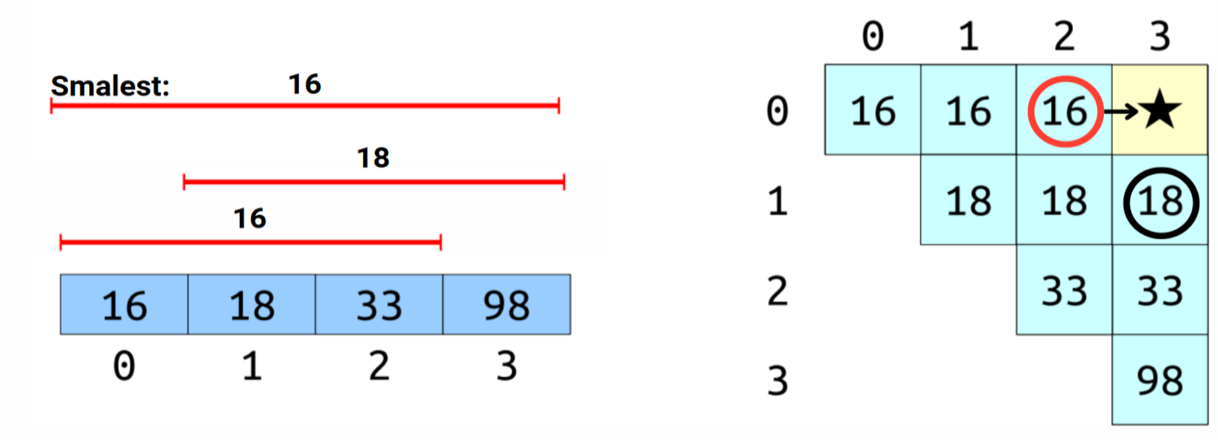
Use Cases:

- **Low Query Frequency:** Suitable when only a few queries are expected, making preprocessing unnecessary.
- **Dynamic Arrays:** Ideal when the array changes frequently, as preprocessing would need to be redone after each modification.

2.2 Precompute All

Description

The "Precompute All" technique is a brute-force preprocessing approach for the Range Minimum Query (RMQ) problem. The idea is to precompute and store the minimum values for **every possible subarray** of the input array. This results in a **n × n** lookup table, where the entry **P[i][j]** represents the minimum element in the range **A[i...j]**. With the precomputed table, each **query** can be **answered** in **constant time O(1)**.

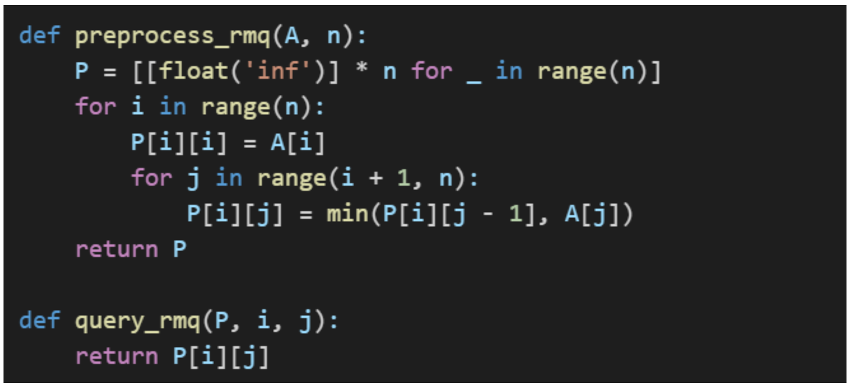


Steps

1. **Initialize a Table:** Create a 2D table **P[n][n]**, where **P[i][j]** will store the **minimum** value in the range **A[i...j]**.
2. **Precompute Ranges:** For each **i ∈ [0, n−1]**:
    - For each **j∈[i,n−1]**:
        - Compute the minimum value in **A[i...j]** using: **P[i][j] = min(P[i][j−1], A[j])** (Alternatively, iterate through the range **A[i...j]** for initialization.)
3. **Query Response:** To answer a query **RMQ(A,i,j)**, return the value in **P[i][j]** directly:

Result = P[i][j]

Pseudocode



```
def preprocess_rmq(A, n):
    P = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        P[i][i] = A[i]
        for j in range(i + 1, n):
            P[i][j] = min(P[i][j - 1], A[j])
    return P

def query_rmq(P, i, j):
    return P[i][j]
```

Theoretical Analysis

1. **Preprocessing Time: O(n^2)**

The preprocessing phase involves filling a 2D table **P[i][j]**. To compute **P[i][j]** for every pair **(i, j),** we iterate through all possible subarrays.

- **Number of Subarrays**:
    1. For an array of size **n**, the number of subarrays can be expressed as: **Total Subarrays =** $\sum_{i=0}^{n-1}(n-i)$
    2. This summation simplifies to: $\sum_{i=0}^{n-1}(n-i)$ **= n * (n+1) / 2**
    3. Therefore, the total number of subarrays is **O(n^2).**

- **Naive Computation of Each Subarray**:
    1. For each subarray **A[i...j],** we iterate through the range **[i,j],** which takes **O(j−i+1)** steps. In the **worst** case, this is **O(n)** for each pair.
    2. Total preprocessing time is then: T_{naive}= $\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1)$
    3. This simplifies to **O(n^3).**

Thus, the naive approach has a preprocessing time of **O(n^3)**, making it impractical for large **n**.

Optimized Preprocessing: Using Dynamic Programming

Using dynamic programming, we reduce the computation time for each subarray by leveraging previously computed results.

- **Dynamic Programming Recurrence**:
    1. To compute **P[i][j],** use**: P[i][j] = min(P[i][j−1], A[j])**
    2. This allows us to compute the minimum in **O(1)** time for each entry in the table.
- **Number of Entries in the Table**:
    1. There are **O(n^2)** entries in the table, as we need to compute **P[i][j]** for all **0 ≤ i ≤ j< n.**

- **Preprocessing Time**:
    1. Since each entry **P[i][j]** is computed in **O(1)**, the total preprocessing time becomes: $\mathbf{T_{optimized}=O(n^2)}$

2. **Query Time: O(1)**
    - Since the minimum value for any range is directly available from the table, no computation is needed during the query phase.
3. **Space Complexity: $\mathbf{O(n^2)}$**
    - The **n×n** table requires quadratic space to store all precomputed range minima.

Advantages

- **Fastest Query Time:** Each query is answered in constant time, **O(1).**
- **Simplicity:** The logic is straightforward, and the method is easy to implement and debug.

Disadvantages

- **High Memory Usage:** The quadratic space requirement makes it impractical for large input sizes.
- **Inefficient for Sparse Queries:** If the number of queries is small, the preprocessing cost outweighs the benefits.
- **Static Arrays Only:** If the array changes, the entire table must be recomputed, making it unsuitable for dynamic scenarios.

Use Case

The "Precompute All" technique is best suited for applications where:

1. Memory is abundant.
2. The input array is static and does not change.
3. There is a very high frequency of range queries requiring instantaneous responses.

2.3 Blocking

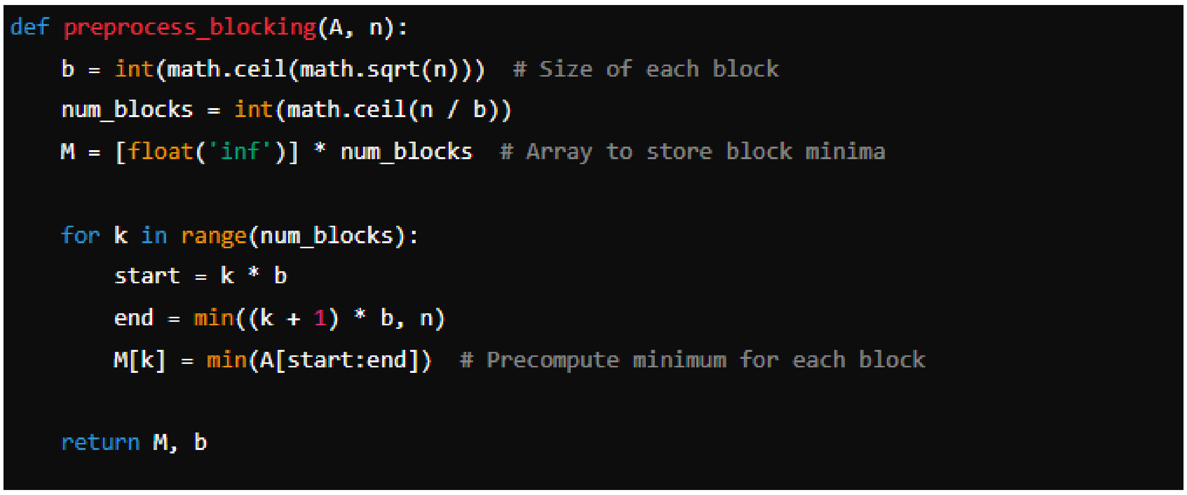
- **Description:** In the Blocking technique for solving the Range Minimum Query (RMQ) problem, the input array is divided into smaller blocks of equal size (except possibly the last block). Each block is individually preprocessed to allow direct and efficient queries within the block. Additionally, the minimum value of each block is precomputed and stored separately to enable efficient inter-block queries.

Steps:

1. **Divide the Array into Blocks**:
    - Split the input array into **$b=\lceil\sqrt{n}\rceil$** blocks, where n is the size of the array. This ensures the number of blocks is **$\lceil n/b \rceil$**.
2. **Precompute Block Minima**:
    - For each block **k**, compute the minimum value within the block and store it in an auxiliary array **M[k]**, where M[k] represents the minimum value of block **k**.
3. **Query Processing**:

- Given a query **RMQ(A,i,j)**:
    - **Case 1: i and j are in the same block**:
        - Perform a linear scan of the elements from i to j within the block to find the minimum.
    - **Case 2: i and j span multiple blocks**:
        - Compute the minimum in three parts:
            1. The partial block containing **i** (from i to the end of the block).
            2. The partial block containing j (from the start of the block to j).
            3. The full blocks between **i** and **j** using the precomputed block minima in **M**.
        - Return the overall minimum from these three parts.

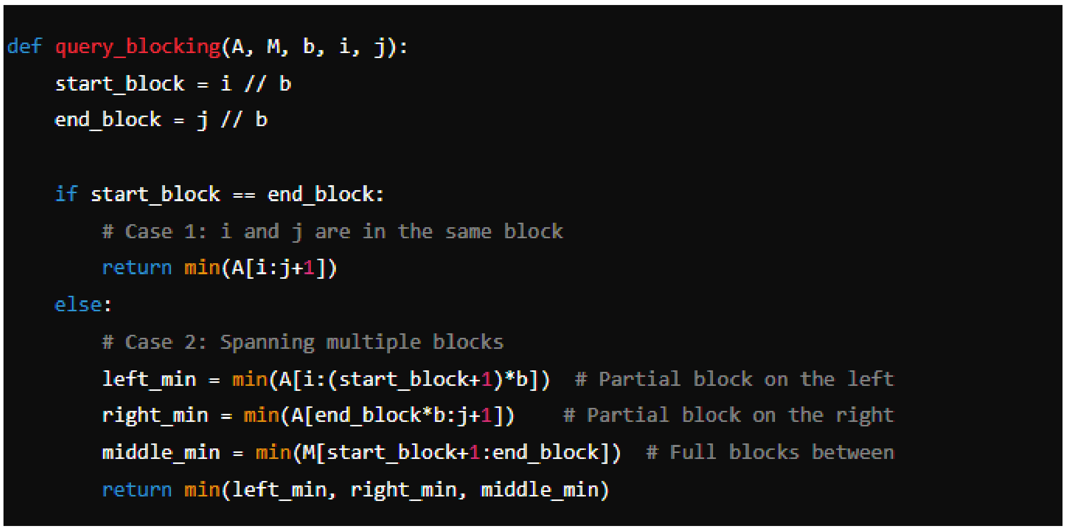
Pseudocode



```python
def preprocess_blocking(A, n):
    b = int(math.ceil(math.sqrt(n)))  # Size of each block
    num_blocks = int(math.ceil(n / b))
    M = [float('inf')] * num_blocks  # Array to store block minima

    for k in range(num_blocks):
        start = k * b
        end = min((k + 1) * b, n)
        M[k] = min(A[start:end])  # Precompute minimum for each block

    return M, b
```



```python
def query_blocking(A, M, b, i, j):
    start_block = i // b
    end_block = j // b

    if start_block == end_block:
        # Case 1: i and j are in the same block
        return min(A[i:j+1])
    else:
        # Case 2: Spanning multiple blocks
        left_min = min(A[i:(start_block+1)*b])  # Partial block on the left
        right_min = min(A[end_block*b:j+1])     # Partial block on the right
        middle_min = min(M[start_block+1:end_block])  # Full blocks between
        return min(left_min, right_min, middle_min)
```

Theoretical Analysis:

1. **Preprocessing Time:**
    - Computing the block minima involves scanning all elements of the array once. For **n** elements, the preprocessing time is: **O(n)**
2. **Query Time:**

Given a query **RMQ(A,i,j)** there are two cases to handle:

- **Partial Blocks**: Handle the elements in the partial blocks:
    1. Left partial block (from i to the end of i's block).
    2. Right partial block (from the beginning of j's block to j).
    3. These two involve a linear scan of at most 2b elements (since each block contains b elements).
    4. Cost: **O(b)**.
- **Full Blocks**: Handle the complete blocks between i's block and j's block:
    1. There are at most n/bn / bn/b such blocks.
    2. The precomputed block minima allow this operation to be completed in **O(n/b)**.
    3. Thus, the total query time T_q is: **Tq=O(b) + O(n/b)**

3. **Total Effort:**
    - The overall effort for a single query and preprocessing is the sum of the preprocessing time and the query time:
    - **$T_{total} = T_p + T_q = O(n) + O(b) + O(n/b)$**
4. **Optimization: Finding Optimal:**
   To minimize **T_{total}**, we need to minimize **O(b+n/b)**. This involves balancing the terms **O(b)** (time for scanning partial blocks) and **O(n/b)** (time for handling full blocks).
    - Define the function:
        1. **f(b) = b + n/b**

        2. Take the derivative of **f(b)** with respect to **b** and set it to zero: $\mathbf{f'(b) = 1 - n/b^2 = 0}$

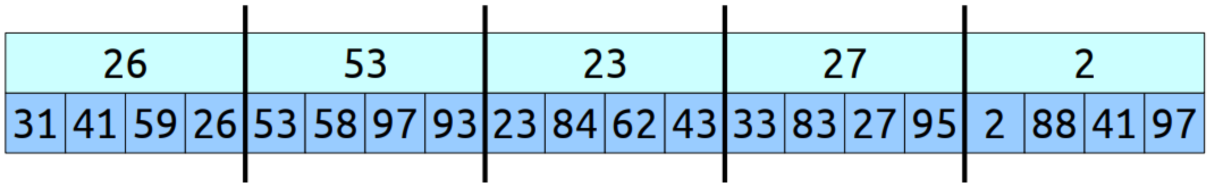
    - Solve for b:

        1. $\mathbf{b^2 = n \Rightarrow b = \sqrt{n}}$

   Thus, the optimal block size **b** is $\mathbf{\sqrt{n}}$.

5. **Space Complexity:**

    - Storing the block minima requires space proportional to the number of blocks, O($\sqrt{\mathbf{n}}$).



Advantages

- Efficient preprocessing (O(n)).
- Relatively fast queries (O($\sqrt{\mathbf{n}}$)) compared to naive methods.
- Memory usage is moderate (O($\sqrt{\mathbf{n}}$)).

Disadvantages

- Query times are slower compared to advanced techniques like Sparse Tables.
- Slightly more complex to implement due to handling partial and full blocks.

Use Case

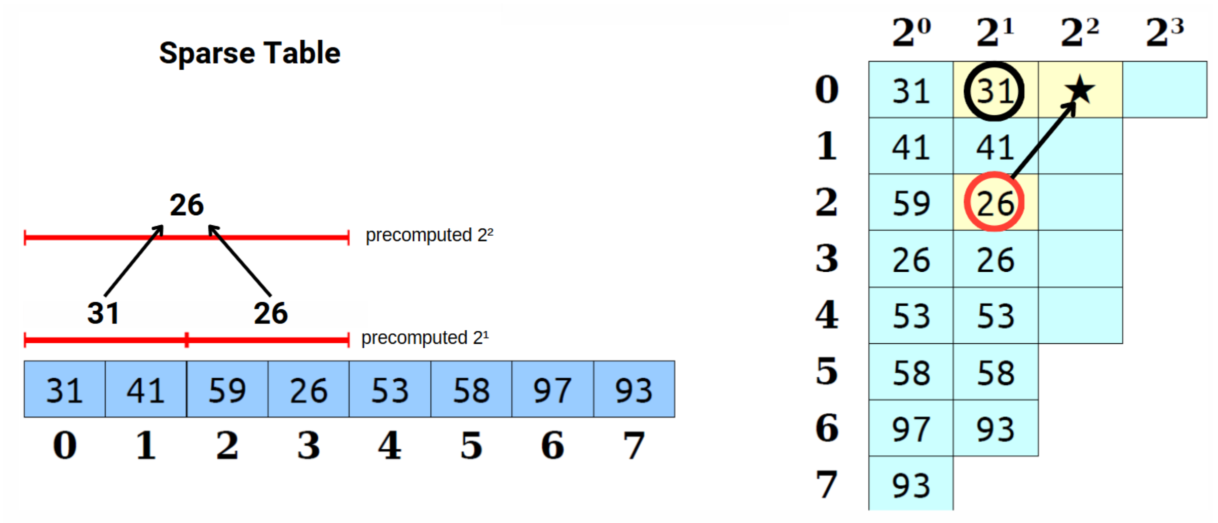
- This approach is ideal for moderate-sized datasets where:

    1. Preprocessing time must remain low.
    2. Queries are frequent but need not be extremely fast.
    3. Memory constraints exist.

2.4 Sparse Table

- **Description:** The **Sparse Table** technique is a highly efficient solution for the Range Minimum Query (RMQ) problem when the array is static (no updates after initialization). It uses a dynamic programming approach to precompute the minimum values for specific overlapping intervals of the array, allowing constant-time query resolution.

- Sparse Table leverages the fact that any range can be represented as the union of two overlapping intervals, which are **powers of 2**. This overlap allows us to reduce storage and preprocessing time compared to the brute-force approach.

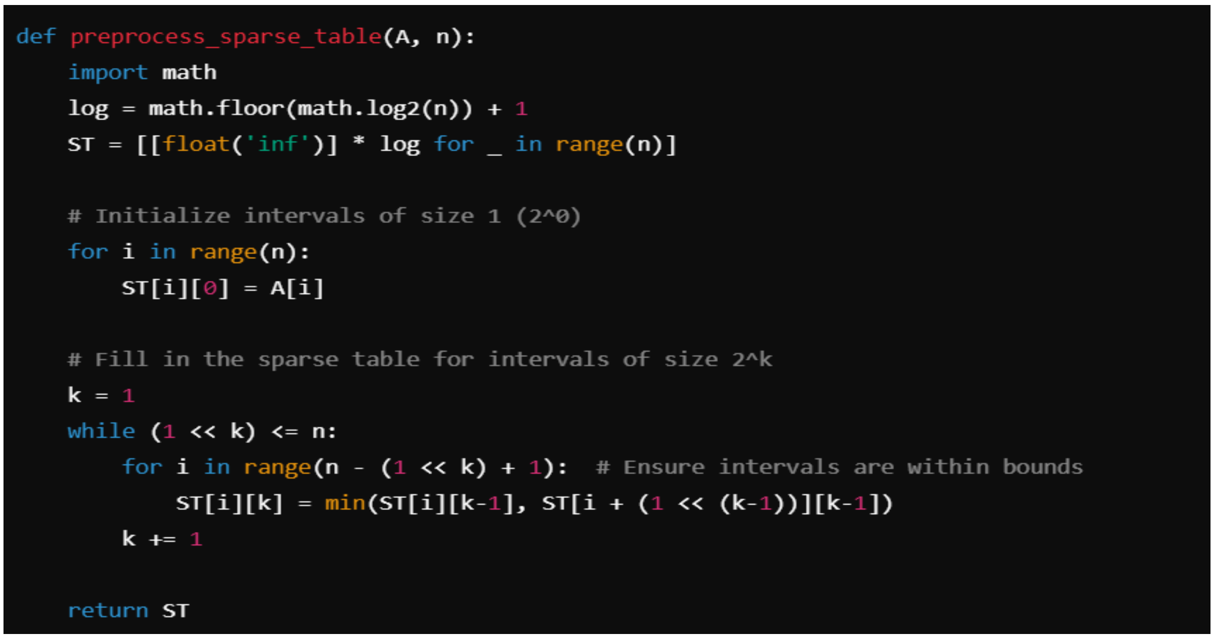


Steps

1. **Precompute Minima for Power-of-Two Ranges:**
    - Divide the array into intervals of size **2^k**, where **k** is the power of 2, and **$1 \le 2^k \le n$**.
    - Precompute the minimum for all possible starting indices iii and interval lengths **2^k**.
2. **Dynamic Programming Table Construction:**
    - Define a table **ST[i][k]**, where **ST[i][k]** represents the minimum value in the range **A[i…i+2k−1]**.
    - Compute **ST[i][k]** recursively: **$ST[i][k] = \min(ST[i][k-1], ST[i+2^{k-1}][k-1])$** Here, **ST[i][k−1]** represents the first half of the range, and **$ST[i+2^{k-1}][k-1]$** represents the second half.
3. **Query Answering:**
    - Any query **RMQ(A,i,j)** can be answered by combining the results of two overlapping intervals: **$RMQ(A,i,j) = \min(ST[i][k], ST[j-2^{k+1}][k])$** where **$k=\lfloor \log_2(j-i+1) \rfloor$**.

Pseudocode

```
def query_sparse_table(ST, i, j):
    import math
    k = math.floor(math.log2(j - i + 1))
    return min(ST[i][k], ST[j - (1 << k) + 1][k])
```



```python
def preprocess_sparse_table(A, n):
    import math
    log = math.floor(math.log2(n)) + 1
    ST = [[float('inf')] * log for _ in range(n)]

    # Initialize intervals of size 1 (2^0)
    for i in range(n):
        ST[i][0] = A[i]

    # Fill in the sparse table for intervals of size 2^k
    k = 1
    while (1 << k) <= n:
        for i in range(n - (1 << k) + 1):  # Ensure intervals are within bounds
            ST[i][k] = min(ST[i][k-1], ST[i + (1 << (k-1))][k-1])
        k += 1

    return ST
```

Theoretical Analysis:

Preprocessing Phase

To construct the sparse table, we precompute the minimum values for intervals of size **2^k**, where **k** is a power of 2. The mathematical formulation for the preprocessing is as follows:

1. **Base Case (k=0)**:
    - For intervals of size $2^0 = 1$, the minimum value in each interval is the element itself: **ST[i][0] = A[i]** for all **$0 \le i < n$**
    - This step requires **O(n)** time since we initialize n entries.
2. **Recursive Case (k>0)**:
    - For intervals of size 2^k, we compute: **ST[i][k] = min(ST[i][k−1], ST[i+2k−1][k−1])**
    - Here:
        - **ST[i][k−1]:** Represents the minimum value in the first half of the interval.
        - **ST[i+2k−1][k−1]**: Represents the minimum value in the second half of the interval.
    - The computation of each **ST[i][k]** entry requires **O(1)** time, and we compute this for all valid iii and k.
3. **Number of Entries**:
    - For each k, the number of valid intervals iii is: **n−2k+1**

- The total number of k-values (powers of 2) is **$\lfloor \log_2(n) \rfloor + 1$**.

4. **Total Time Complexity**:
   - Summing over all k: **T_p=** $\sum_{k=0}^{log2(n)}(n - 2\text{^}k + 1)$
   - This simplifies to: T_p=**O(nlogn)**

Query Phase

To answer a query **RMQ(A,i,j),** we use the property of overlapping intervals:

1. **Finding the Interval Size**:
   - Compute **$k = \lfloor \log_2(j-i+1) \rfloor$,** which determines the largest power-of-2 interval size that fits within **[i,j].**
2. **Combining Two Overlapping Intervals**:
   - The range**[i,j]** can be divided into two overlapping intervals of size 2^k: **RMQ(A,i,j) = min(ST[i][k], ST[j−2k+1][k])**
   - Both intervals **[i, i+2^k−1]** and **[j−2^k+1,j]** are precomputed in **ST[i][k]** and **ST[j−2k+1][k].**
3. **Time Complexity**:
   - The query involves a single min operation, which is **O(1).**

Space Complexity

1. The sparse table stores **ST[i][k]** for all i and k, requiring: **S=O(nlogn)**

Advantages:

- Efficient preprocessing and query times.
- Lower memory requirements compared to the "Precompute All" approach.

Disadvantages:

- Slightly more complex implementation.

Use Case:

- Ideal for applications where memory constraints exist, and query times need to be very fast.

3. Experimental Design

The goal of the experimental design is to empirically evaluate and compare the performance of four Range Minimum Query (RMQ) algorithms:

1. **Precompute All**
2. **Sparse Table**
3. **Blocking**
4. **Precompute None**

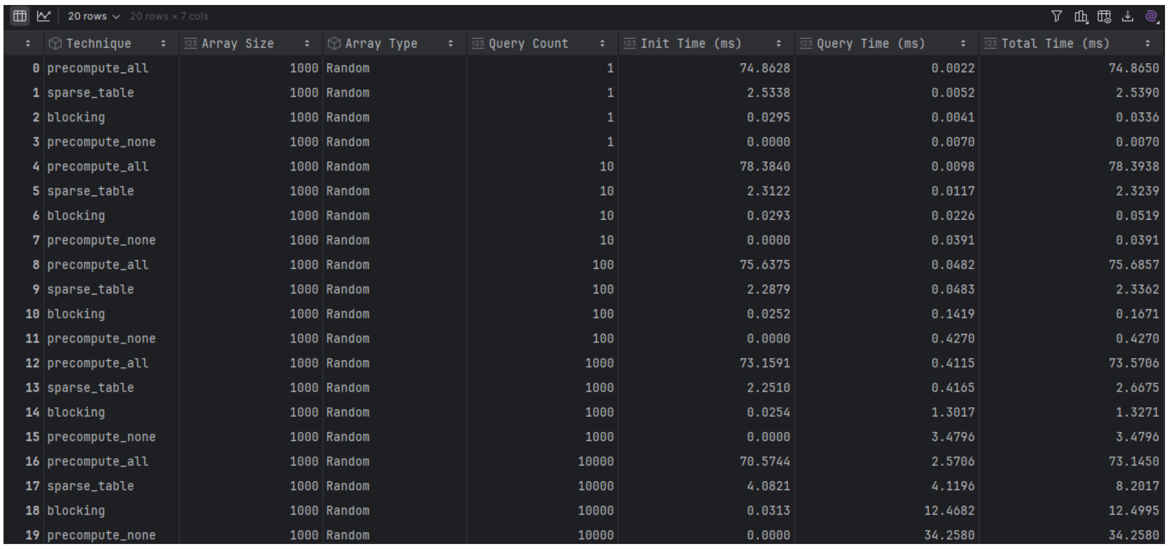
Each algorithm has different preprocessing and query execution strategies, leading to distinct theoretical time complexities. We conduct experiments to measure the **initialization time**, **query execution time**, and **total time (initialization + query)** across varying parameters to validate the algorithms' time complexities empirically.

Experiments and Implementation

We conducted three main experiments:

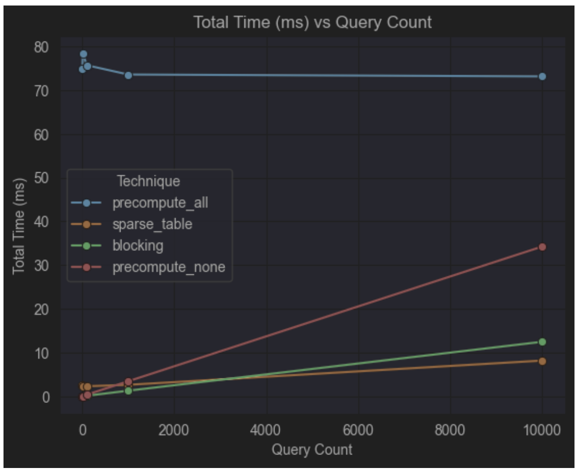
1. Varying Query Count (Constant Array Size)

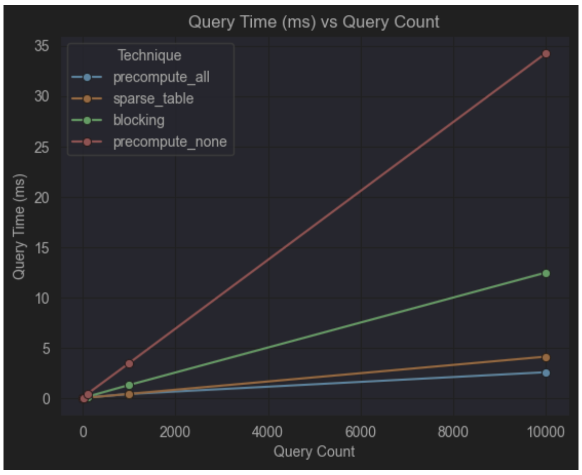
- **Goal**: Analyze how the number of queries affects the total execution time for each algorithm.
- **Design**:
    - Use a constant array size (e.g., 1000 elements).
    - Vary the query count (e.g., 1, 10, 100, 1000, 10000).
    - Measure the **initialization** time, **query** time, and **total** time for each algorithm.
- **Implementation**:
    - Generate a random array of size 1000.
    - For each query count, generate **random query ranges** [i,j].
    - Execute all algorithms and record their performance metrics.
- **Experiment Outputs:**



20 rows 20 rows × 7 cols

	Technique	Array Size	Array Type	Query Count	Init Time (ms)	Query Time (ms)	Total Time (ms)
0	precompute_all	1000	Random	1	74.8628	0.0022	74.8650
1	sparse_table	1000	Random	1	2.5338	0.0052	2.5390
2	blocking	1000	Random	1	0.0295	0.0041	0.0336
3	precompute_none	1000	Random	1	0.0000	0.0070	0.0070
4	precompute_all	1000	Random	10	78.3840	0.0098	78.3938
5	sparse_table	1000	Random	10	2.3122	0.0117	2.3239
6	blocking	1000	Random	10	0.0293	0.0226	0.0519
7	precompute_none	1000	Random	10	0.0000	0.0391	0.0391
8	precompute_all	1000	Random	100	75.6375	0.0482	75.6857
9	sparse_table	1000	Random	100	2.2879	0.0483	2.3362
10	blocking	1000	Random	100	0.0252	0.1419	0.1671
11	precompute_none	1000	Random	100	0.0000	0.4270	0.4270
12	precompute_all	1000	Random	1000	73.1591	0.4115	73.5706
13	sparse_table	1000	Random	1000	2.2510	0.4165	2.6675
14	blocking	1000	Random	1000	0.0254	1.3017	1.3271
15	precompute_none	1000	Random	1000	0.0000	3.4796	3.4796
16	precompute_all	1000	Random	10000	70.5744	2.5706	73.1450
17	sparse_table	1000	Random	10000	4.0821	4.1196	8.2017
18	blocking	1000	Random	10000	0.0313	12.4682	12.4995
19	precompute_none	1000	Random	10000	0.0000	34.2580	34.2580





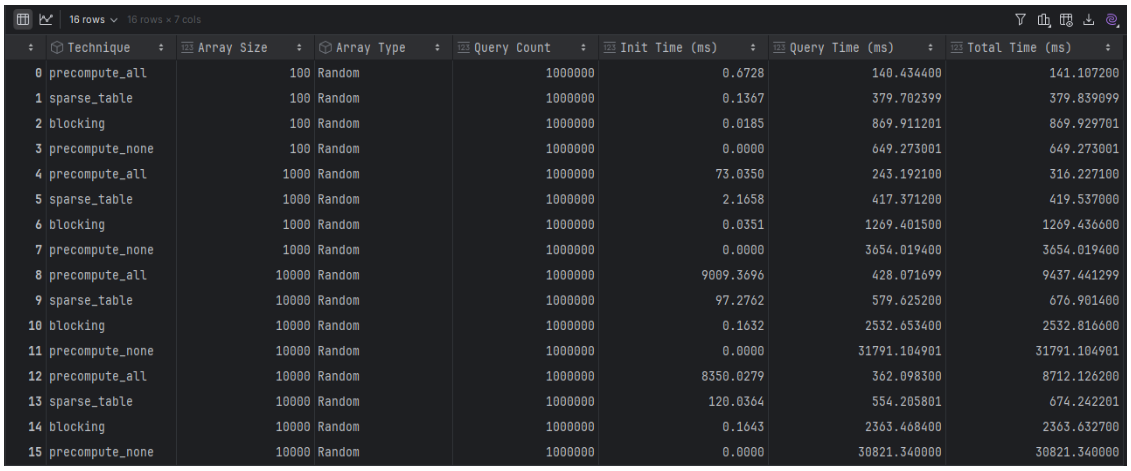
- **Expected Relation to Time Complexity**:
  - **Precompute All**: Query time is **O(1)**, so total time depends mainly on preprocessing.
  - **Sparse Table**: Query time is **O(1),** so total time grows linearly with query count after preprocessing.
  - **Blocking**: Query time is **O(√n),** leading to higher total time with increased queries.
  - **Precompute None**: Query time is **O(n),** causing linear growth of total time with query count.

---

2. Varying Array Size (Constant Query Count)

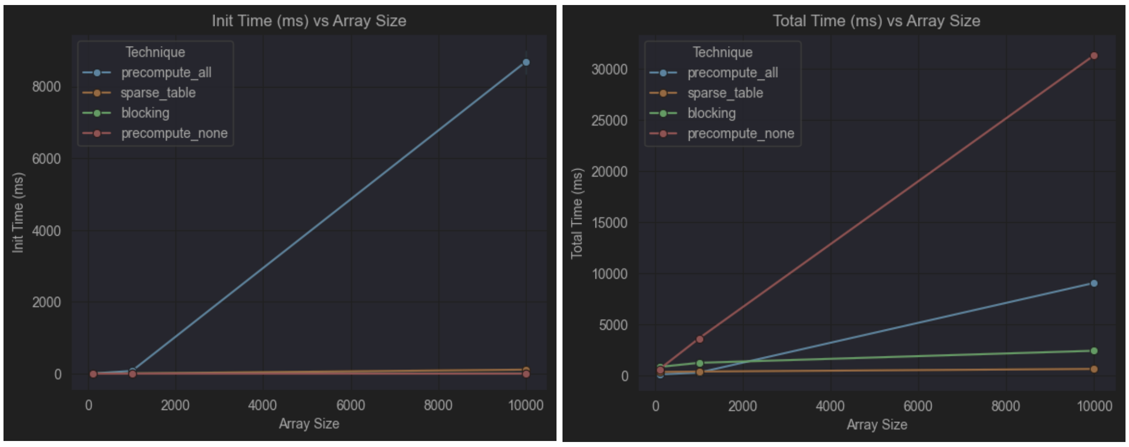
- **Goal:** Examine how the array size impacts execution time for each algorithm.

- **Design**:
    - Use a **constant query count** (e.g., 100 queries).
    - Vary the array size (e.g., 100, 1000, 10000, 100000).
    - Measure the initialization time, query time, and total time for each algorithm.
- **Implementation**:
    - Generate random arrays of different sizes.
    - Execute 100 queries with randomly generated query ranges [i,j].
    - Record the initialization, query, and total time for each algorithm.
- **Experiment Outputs:**



16 rows   16 rows × 7 cols

	Technique	Array Size	Array Type	Query Count	Init Time (ms)	Query Time (ms)	Total Time (ms)
0	precompute_all	100	Random	1000000	0.6728	140.434400	141.107200
1	sparse_table	100	Random	1000000	0.1367	379.702399	379.839099
2	blocking	100	Random	1000000	0.0185	869.911201	869.929701
3	precompute_none	100	Random	1000000	0.0000	649.273001	649.273001
4	precompute_all	1000	Random	1000000	73.0350	243.192100	316.227100
5	sparse_table	1000	Random	1000000	2.1658	417.371200	419.537000
6	blocking	1000	Random	1000000	0.0351	1269.401500	1269.436600
7	precompute_none	1000	Random	1000000	0.0000	3654.019400	3654.019400
8	precompute_all	10000	Random	1000000	9009.3696	428.071699	9437.441299
9	sparse_table	10000	Random	1000000	97.2762	579.625200	676.901400
10	blocking	10000	Random	1000000	0.1632	2532.653400	2532.816600
11	precompute_none	10000	Random	1000000	0.0000	31791.104901	31791.104901
12	precompute_all	10000	Random	1000000	8350.0279	362.098300	8712.126200
13	sparse_table	10000	Random	1000000	120.0364	554.205801	674.242201
14	blocking	10000	Random	1000000	0.1643	2363.468400	2363.632700
15	precompute_none	10000	Random	1000000	0.0000	30821.340000	30821.340000



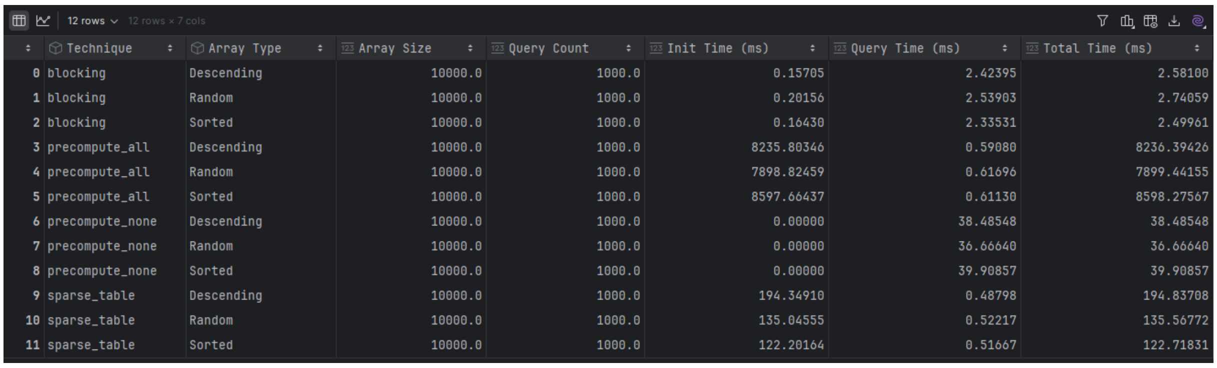
- **Expected Relation to Time Complexity**:
    - **Precompute All**: Initialization time dominates with **O(n^2),** making it infeasible for large arrays.
    - **Sparse Table**: Preprocessing time grows as **O(nlogn);** total time remains efficient due to query time.

    - **Blocking**: Preprocessing is **O(n),** but query time leading to a moderate increase in total time.
    - **Precompute None**: Sinse there is no preprocessing, both query time and total time grow linearly **O(n)** with array size.

---

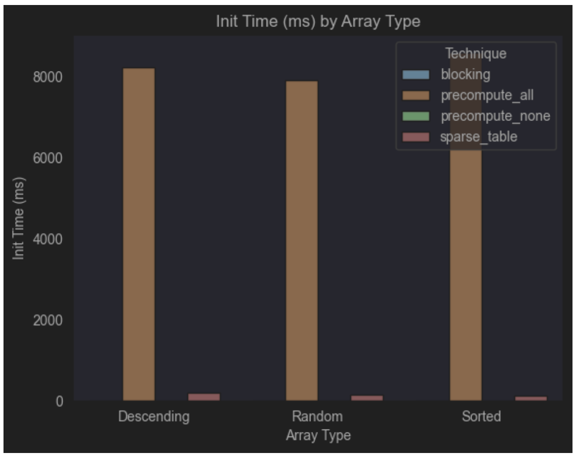
3. Varying Array Types (Constant Array Size and Query Count)

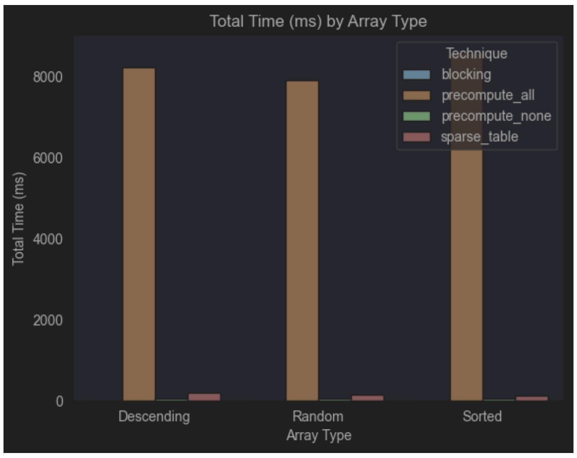
- **Goal**: Evaluate how array properties (e.g., sorted, random, descending) influence performance.
- **Design**:
    - Use a constant array size (e.g., 10,000 elements).
    - Use a constant query count (e.g., 1000 queries).
    - Measure the initialization time, query time, and total time for each algorithm on:
        - Random Array
        - Sorted Array
        - Descending Array
- **Implementation**:
    - **Generate** arrays of different types (random, sorted, descending).
    - **Execute** 1000 queries with randomly generated query ranges [i,j].
    - **Record** the initialization, query, and total time for each algorithm.
    - **Repeat** the process 10 times and get the average results.
- **Experiment Outputs:**



12 rows 12 rows × 7 cols

	Technique	Array Type	Array Size	Query Count	Init Time (ms)	Query Time (ms)	Total Time (ms)
0	blocking	Descending	10000.0	1000.0	0.15705	2.42395	2.58100
1	blocking	Random	10000.0	1000.0	0.20156	2.53903	2.74059
2	blocking	Sorted	10000.0	1000.0	0.16430	2.33531	2.49961
3	precompute_all	Descending	10000.0	1000.0	8235.80346	0.59080	8236.39426
4	precompute_all	Random	10000.0	1000.0	7898.82459	0.61696	7899.44155
5	precompute_all	Sorted	10000.0	1000.0	8597.66437	0.61130	8598.27567
6	precompute_none	Descending	10000.0	1000.0	0.00000	38.48548	38.48548
7	precompute_none	Random	10000.0	1000.0	0.00000	36.66640	36.66640
8	precompute_none	Sorted	10000.0	1000.0	0.00000	39.90857	39.90857
9	sparse_table	Descending	10000.0	1000.0	194.34910	0.48798	194.83708
10	sparse_table	Random	10000.0	1000.0	135.04555	0.52217	135.56772
11	sparse_table	Sorted	10000.0	1000.0	122.20164	0.51667	122.71831





- **Expected Relation to Time Complexity**:
  - Array type does not affect preprocessing or query complexities because all algorithms rely on indices rather than data patterns.
  - There are slightly differences in preprocess performance. This is likely be due to caching or localized data access patterns.

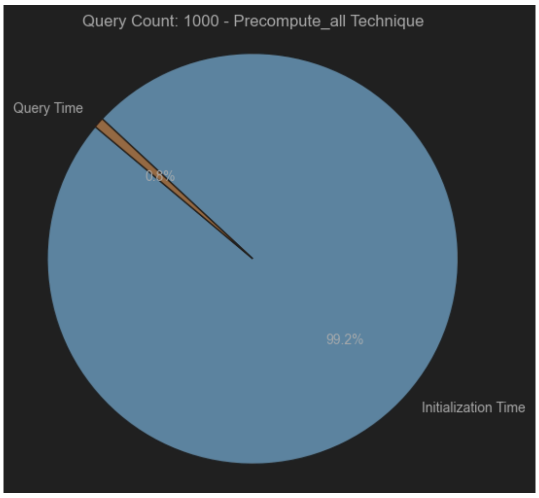
---

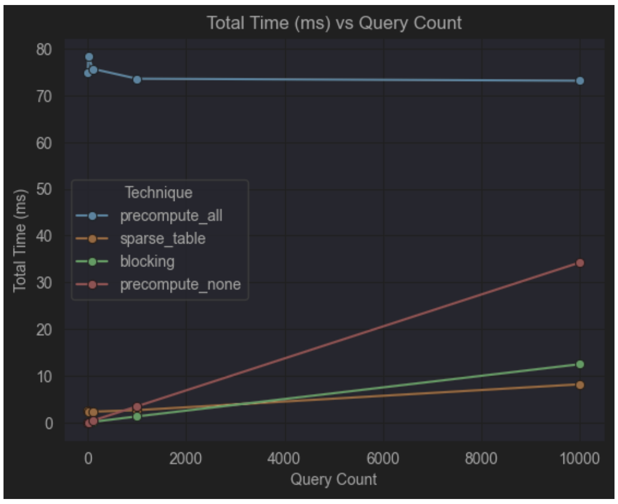
4. Results and Analysis

4.1 Varying Query Count Experiment

1. Precompute All

- **Detailed Graphs:**





- **Initialization Time**:
  - Initialization time is **consistent** across **all query counts** (~75–78 ms), as it **depends only on the array size** (O(n²)).
- **Query Time**:

- Query execution time is **negligible** due to its **O(1)** query complexity (e.g., ~0.002 ms for 1 query and ~0.048 ms for 100 queries).

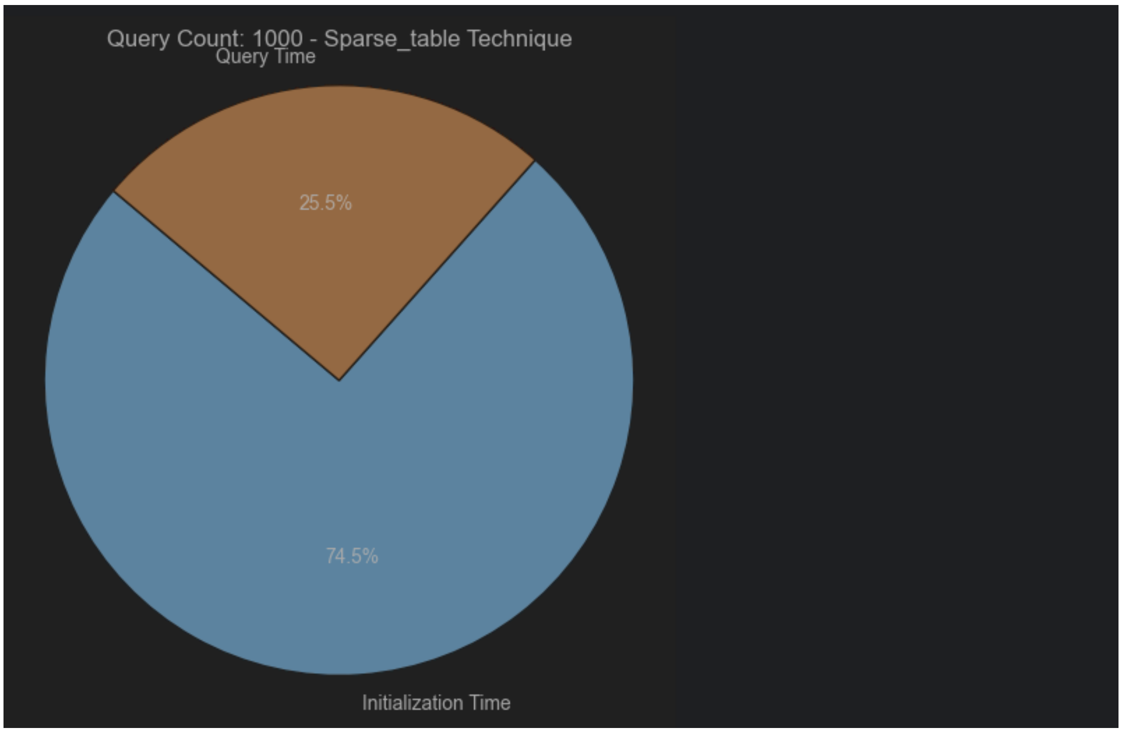
- **Total Time**:
  - Total time is almost **entirely dominated by initialization**, ranging from 74.87 ms (1 query) to 78.39 ms (10 queries).

- **Inference**:
  - **Precompute All** is **efficient for** scenarios with extremely **frequent queries,** as the high preprocessing cost is amortized across many queries.

  - It is inefficient for smaller query counts due to its **quadratic** preprocessing **cost**.

---

2. **Sparse Table**
   - **Detailed Graphs:**



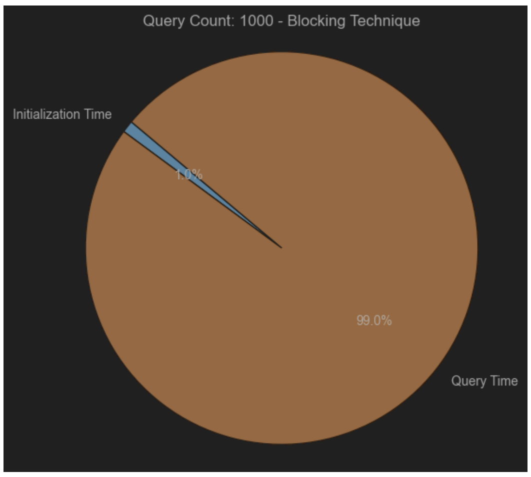
   - **Initialization Time**:
     - Sparse Table's initialization time remains low (~2.3 ms) for all query counts.

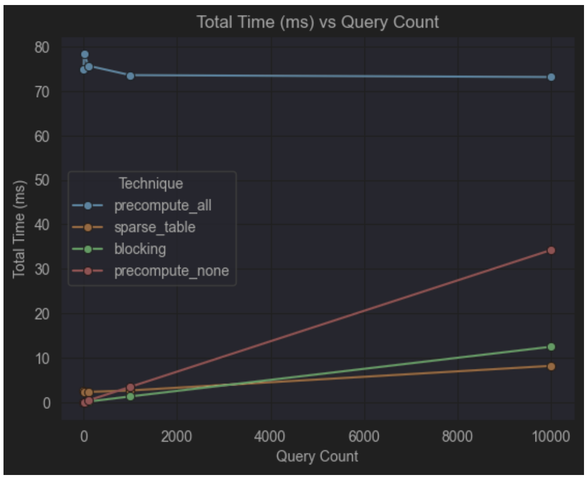
- **Query Time**:
  - Query execution time is **proportional** to the **number of queries**, **increasing slightly** with query **count** (e.g., ~0.005 ms for 1 query and ~0.048 ms for 100 queries).
- **Total Time**:
  - Total time **grows modestly** with query count, ranging from 2.54 ms (1 query) to 2.34 ms (100 queries).
- **Inference**:
  - **Sparse Table** is an excellent choice for scenarios **with frequent queries** and **moderate preprocessing** needs.
  - Its performance stable and **scales well** with **increasing query counts**.

---

3. Blocking

- **Detailed Graphs:**





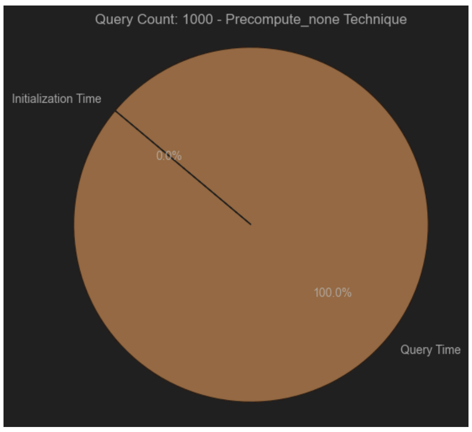
- **Initialization Time**:
  - Initialization time is **minimal (**~0.03 ms) across all query counts due to its **O(n)** preprocessing complexity.
- **Query Time**:
  - Query execution time **grows moderately** with the number of **queries** (e.g., ~0.004 ms for 1 query and ~0.022 ms for 10 queries), reflecting its **O(√n)** query complexity.

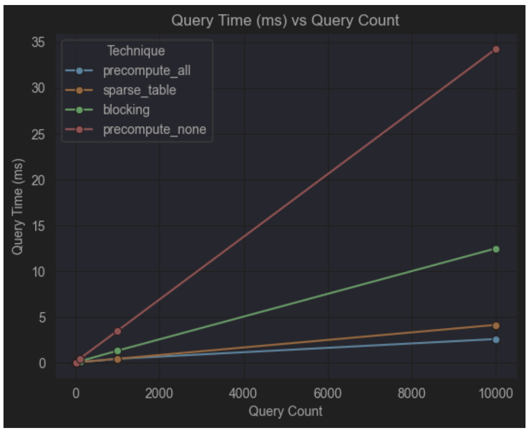
- **Total Time**:
  - Total time remains low for smaller query counts but increases proportionally to query count (e.g., 0.034 ms for 1 query and 0.052 ms for 10 queries).
- **Inference**:
  - **Blocking** is a suitable choice for scenarios requiring minimal preprocessing, though its query time is slower compared to Sparse Table or Precompute All.

---

4. **Precompute None**

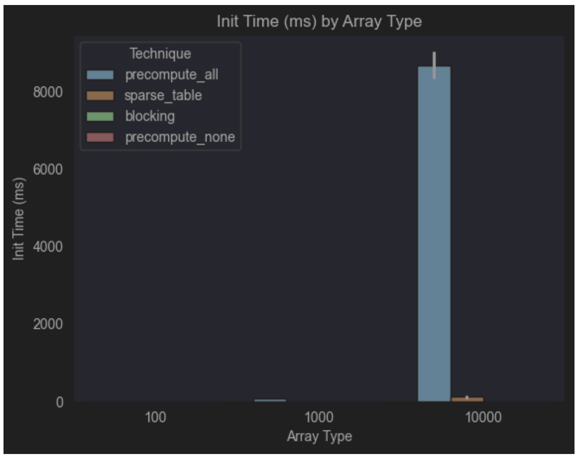
- **Detailed Graphs:**

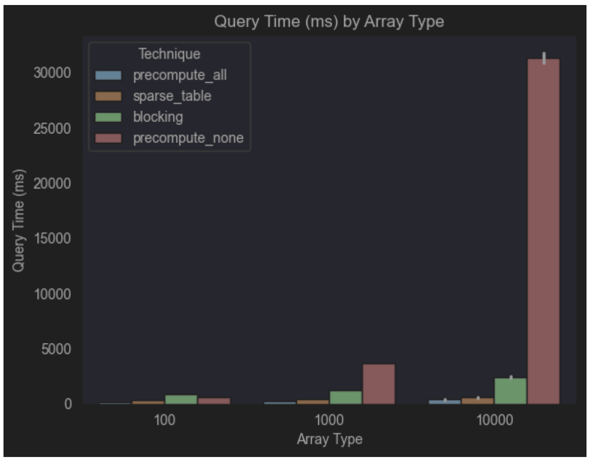


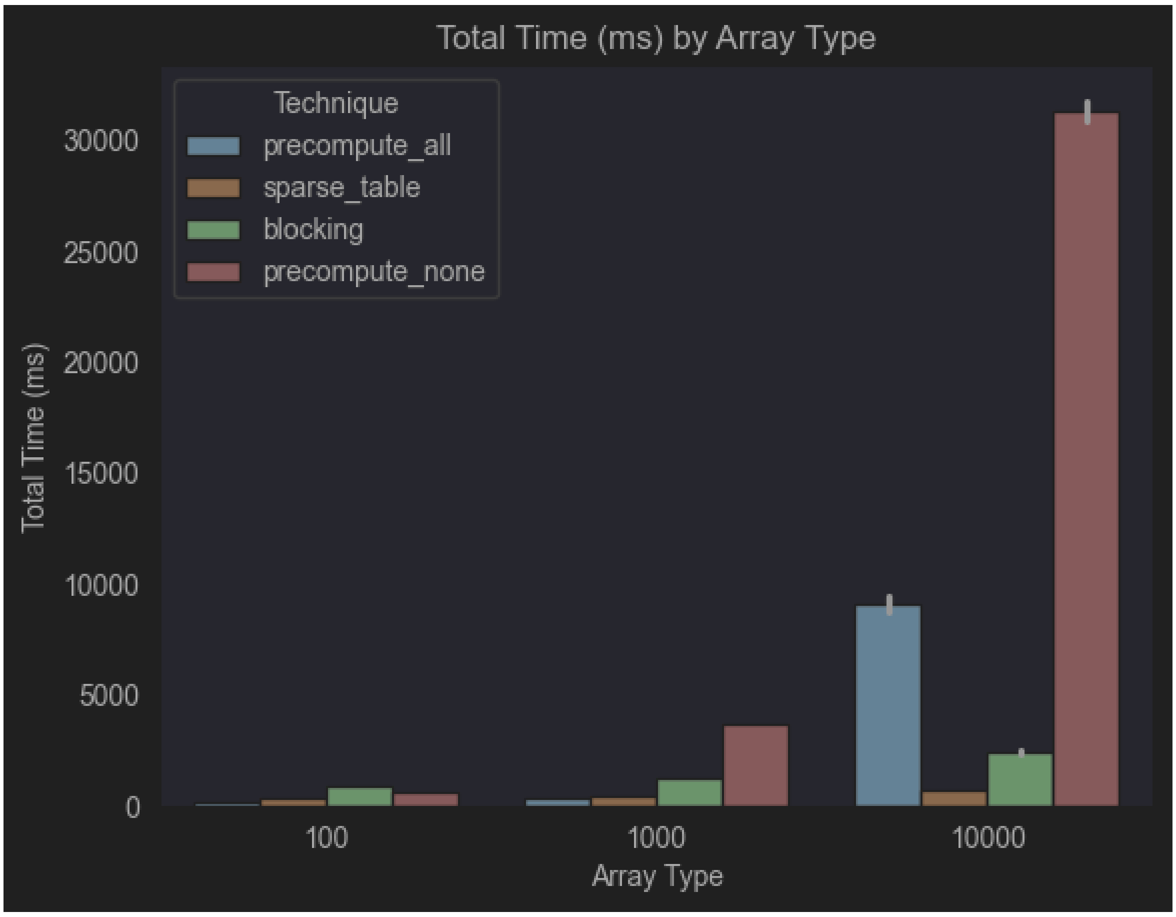


- **Initialization Time**:
  - No preprocessing is performed, so initialization time is **consistently zero.**
- **Query Time**:
  - Query execution time **grows linearly** with the number of queries (e.g., ~0.007 ms for 1 query and ~0.039 ms for 10 queries), reflecting its **O(n)** query complexity.
- **Total Time**:
  - Total time matches query time, increasing proportionally to the number of queries.
- **Inference**:
  - **Precompute None** is highly inefficient for frequent queries due to its lack of preprocessing, making it suitable only for dynamic datasets with very few queries.

4.2 Varying Array Size Experiment







1. Precompute All

- **Initialization Time**:
  - For array size 100, the preprocessing time is **negligible (~0.67 ms).**

    - For array size 1000, the preprocessing time **increases significantly** to **73 ms.**
    - For array size 10,000, the **preprocessing** time **skyrockets** to **9009 ms, confirming its O(n^2)** time complexity.
- **Query Time**:
    - Query time is consistently **low** (around 0.14–0.43 ms) due to its **O(1)** query complexity.
- **Total Time**:
    - Total time is **heavily dominated** by the **preprocessing** time, reaching 9437 ms for the largest array size.
- **Inference**:
    - **Precompute All** becomes **impractical** for **large** arrays due to its quadratic preprocessing cost.
    - **Effective** only for **small arrays** or scenarios requiring **extremely frequent queries**.

2. Sparse Table

- **Initialization Time**:
    - The preprocessing time is low for smaller arrays (0.13 ms for size 100) and grows moderately with array size (O(nlog⁡n)O(n \log n)O(nlogn)), reaching 97 ms for size 10,000.
- **Query Time**:
    - Query time **increases** slightly **with** array **size**, ranging from 379 ms (size 100) to 579 ms (size 10,000). This **reflects** its **O(1)** query complexity.
- **Total Time**:
    - Total time **grows steadily** but **remains reasonable** compared to Precompute All, reaching **676 ms** for the **largest** array size.
- **Inference**:
    - **Sparse Table** balances preprocessing and query time well, making it **suitable for larger arrays with frequent queries.**

3. Blocking

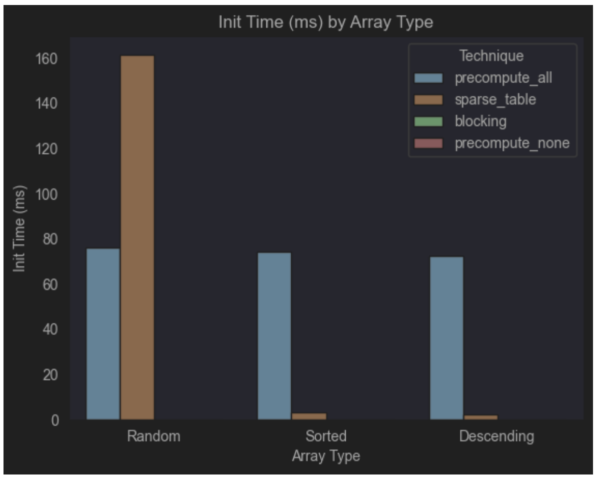
- **Initialization Time**:

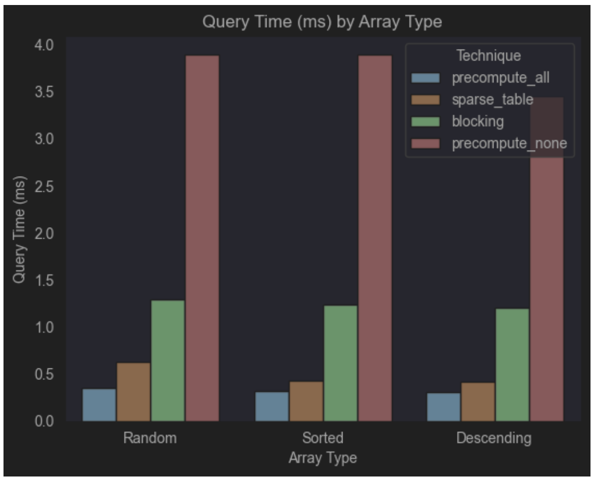
    - Preprocessing time is **minimal** for all array **sizes** (~0.018–0.035 ms), reflecting its **linear preprocessing** complexity **(O(n))**.
- **Query Time**:
    - Query time **grows significantly** with **array size**, from 869 ms (size 100) to 1269 ms (size 10,000), **confirming** its **O(√n)** query **complexity**.
- **Total Time**:
    - Total time is **dominated by query** execution, reaching 1269 ms for the **largest** array **size**.
- **Inference**:
    - **Blocking** offers **minimal preprocessing** cost but **slower query** times compared to Sparse Table and Precompute All.
    - **Suitable for smaller arrays** or scenarios where preprocessing must remain minimal.

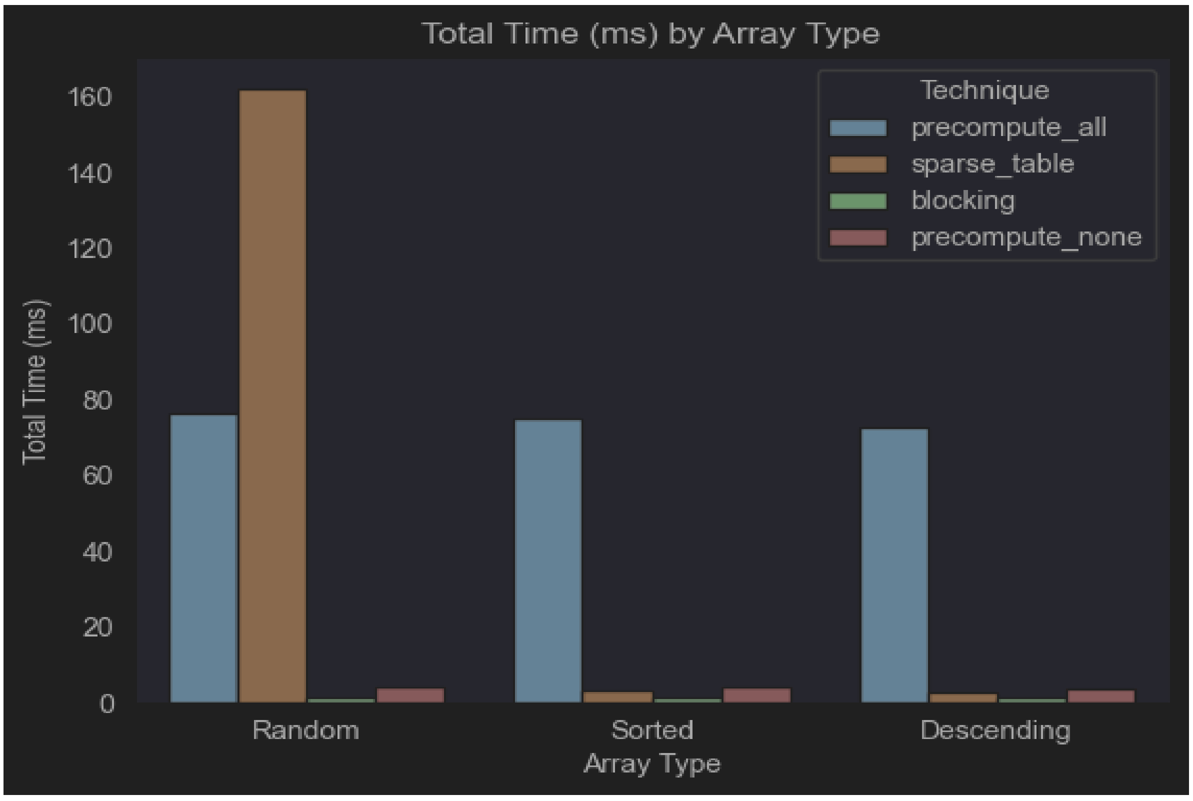
4. Precompute None

- **Initialization Time**:
    - No preprocessing is performed, so initialization time is **always zero.**
- **Query Time**:
    - Query time **grows significantly** with array **size** due to its **O(n)** complexity, from 649 ms (size 100) to 3654 ms (size 10,000).
- **Total Time**:
    - Total time matches query time, reaching 3654 ms for the largest array size.
- **Inference**:
    - **Precompute None** is **inefficient** for **large** arrays or **frequent** queries due to its linear query complexity.
    - **Suitable only for dynamic** datasets where preprocessing is infeasible.

4.3 Varying Array Types Experiment







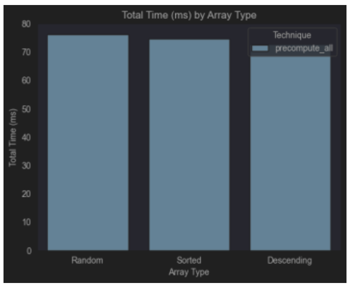
1. Blocking

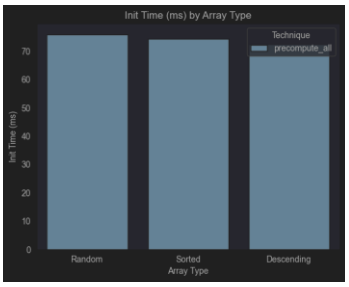
- **Initialization Time**: The **preprocessing** time is **negligible** across **all** array types (less than 0.2 ms).
- **Query Time**:
  - The query time is **consistent** and averages around **2.4–2.5 ms** across all array types.
  - **Random** arrays take **slightly longer** (2.54 ms), while **Sorted** arrays are the **fastest** (2.33 ms).

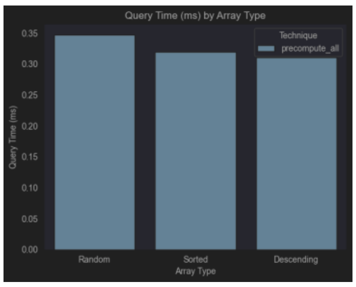
- **Total Time**: The total time is largely dominated by query execution due to minimal preprocessing costs.
- **Inference**:
    - Blocking is efficient for scenarios with minimal preprocessing needs and moderately fast query execution (O(n)).
    - The slight variation in query times could result from memory access patterns, where Sorted arrays may benefit from better cache locality.

2. Precompute All

- **Detailed Graphs:**







- **Initialization Time**:
    - Extremely **high preprocessing** time across **all** array types (around 8,000–8,600 ms).
    - **Sorted** arrays take the **longest preprocessing** time **(8,598 ms)**, followed by Descending **(8,236 ms)** and Random **(7,899 ms).**
- **Query Time**:
    - **Query** execution is the **fastest among all** techniques, averaging less than **0.6 ms** across **all array types** due to its **O(1)** query complexity.
- **Total Time**:
    - Total time is almost entirely **dominated by initialization** costs, making this approach **impractical** for dynamic or **large datasets**.
- **Inference**:
    - Precompute All is effective only in scenarios where the dataset is static, and queries are extremely frequent, as the high initialization cost outweighs its benefits for smaller numbers of queries.

3. Precompute None

- **Initialization Time**: **Zero preprocessing** time due to its on-demand query execution.

- **Query Time**:
    - Query time is **significantly higher** than **other techniques**, averaging between **36.6–39.9** ms **across array types**.
    - **Sorted** arrays take the **longest** time **(39.91 ms),** while **Random** arrays are slightly **faster** (36.67 ms).
- **Total Time**: **Dominated** entirely by **query** execution costs.

4. Sparse Table

- **Initialization Time**:
    - Sparse Table has a **moderate initialization** cost, averaging **194 ms** across **array types**.
- **Query Time**:
    - The query time is **exceptionally fast**, around **0.48 ms** across **all array types.**
- **Total Time**:
    - Total time is **well-balanced**, making Sparse Table **efficient** for both **preprocessing** and **query** execution.

5. Conclusion

The comparative analysis of the four Range Minimum Query (RMQ) algorithms—**Precompute All**, **Sparse Table**, **Blocking**, and **Precompute None**—yielded valuable insights into their theoretical and empirical performance. Through extensive experiments, we evaluated their behavior under varying conditions such as **query counts**, **array sizes**, and **data distributions**. Here are the key findings and general inferences from the analysis:

Key Insights

1. **Trade-Offs Between Preprocessing and Query Efficiency**:
    - Algorithms like **Precompute All** and **Sparse Table** excel at minimizing query times, as they preprocess the array extensively to achieve **O(1)** query complexity. However, this comes at the cost of higher preprocessing times.
    - Conversely, algorithms such as **Precompute None** and **Blocking** favor minimal preprocessing but sacrifice query efficiency.
2. **Scalability with Array Size**:

- **Sparse Table** consistently demonstrated its scalability with increasing array sizes due to its **O(nlogn)** preprocessing complexity and constant **O(1)** query time.
- **Precompute All** struggled with larger arrays, as its preprocessing cost grows quadratically **($O(n^2)$)**, making it impractical for large datasets.
- **Blocking** provided a reasonable trade-off, with linear preprocessing time **O(n)** but slower query times **($O(\sqrt{n})$)**.

3. **Behavior Under Varying Query Counts**:
   - For high query counts, **Sparse Table** and **Precompute All** performed the best, as their preprocessing costs were amortized over a large number of queries.
   - **Precompute None**, with no preprocessing, exhibited linear growth in total time as query count increased, making it unsuitable for frequent queries.
   - **Blocking** remained competitive for smaller query counts but lagged behind Sparse Table for higher query volumes.
4. **Effect of Array Types**:
   - The results showed negligible differences in performance across random, sorted, and descending arrays for all algorithms, confirming that the theoretical time complexities are independent of data distribution.
   - Minor variations in query times for **Blocking** were observed, likely due to memory access patterns.

General Inferences

1. **Sparse Table as the Best Overall Technique**:
   - **Sparse Table** emerged as the most **balanced and practical** algorithm for a wide range of scenarios, offering efficient preprocessing and fast query execution. It is ideal for large datasets and applications with frequent queries.
2. **Use Cases for Each Algorithm**:
   - **Precompute All**: Suitable for static datasets where preprocessing can be performed once, and queries are extremely frequent.
   - **Blocking**: A good option for dynamic datasets with smaller query volumes, where preprocessing time must remain minimal.

- **Precompute None**: The only feasible choice for fully dynamic datasets, but its linear query cost limits its usability for high query counts.

3. **Empirical Validation of Theoretical Complexities**:
    - The experiments closely aligned with the expected theoretical time complexities, providing empirical evidence for the mathematical analyses.
    - The results emphasized the importance of understanding the trade-offs between preprocessing and query execution for selecting the appropriate algorithm based on use case.

Final Thoughts

This study highlights the diversity of RMQ techniques and their applicability to different scenarios. While **Sparse Table** is the most versatile and efficient algorithm for static datasets with frequent queries, the other techniques offer viable alternatives depending on specific requirements, such as dynamic datasets or minimal preprocessing constraints. These findings underscore the importance of algorithmic design in optimizing performance for specific problem domains.

Anyone can find the Java source code and Python Jupyter Notebook in this Github repository: https://github.com/EmirhanSyl/RangeMinimumQuery