Range Minimum Query: A Friendly Step-by-Step Tutorial

A Comprehensive Guide to RMQ Algorithms

2025

Contents

What is Range Minimum Query (RMQ)?

Imagine you have a list of numbers, and someone keeps asking you: "What's the smallest number between position 3 and position 7?" That's exactly what RMQ solves!

Example Array:

```
Index: 0  1  2  3  4  5  6  7
Value: 5  2  4  7  1  3  6  8
```

Query(2, 5) = "What's the minimum between index 2 and 5?" - Look at values: [4, 7, 1, 3] - Answer: 1 (at index 4)

Now, let's explore 5 different ways to solve this problem, from simple to sophisticated!

Algorithm 1: Naive Approach (The Straightforward Way)

How It Works

Just look at every element in the range and find the minimum. It's like reading through a list with your finger!

Step-by-Step Example

Array: [5, 2, 4, 7, 1, 3, 6, 8]

Query(2, 5):

```
Step 1: Look at index 2 → value is 4, min = 4
Step 2: Look at index 3 → value is 7, min = 4 (no change)
Step 3: Look at index 4 → value is 1, min = 1 (new minimum!)
Step 4: Look at index 5 → value is 3, min = 1 (no change)
Answer: 1
```

Visual Representation

```
Query(2, 5):
[5, 2, |4, 7, 1, 3|, 6, 8]
        ↑  ↑  ↑  ↑
        Check each one
        Return: 1
```

Time Complexity Deep Dive

Why is Preprocessing O(1)?

- We literally do nothing! Just store the array as-is.
- No computation, no extra data structures.
- Time taken: constant, regardless of array size.

Why is Query Time O(n)? Let's count the operations:

```
Query(L, R):
1. Initialize min = array[L]     → 1 operation
2. For each element from L+1 to R:
   - Compare with current min    → (R-L) comparisons
   - Update min if needed        → up to (R-L) assignments
Total: 2(R-L) + 1 operations
```

Worst case: Query(0, n-1) checks all n elements → O(n) **Best case:** Query(i, i) checks 1 element → O(1) **Average case:** Query covers n/2 elements → O(n)

Space Complexity: O(n)

- Original array: n elements × 4 bytes (for int) = 4n bytes
- No additional structures needed
- Total space: O(n)

Real-world Performance For an array of 1,000,000 elements: - Preprocessing: 0 microseconds - Query (worst case): ~1,000 microseconds (1 ms) - Query (average): ~500 microseconds

When to Use: - Queries are rare (< 100 queries total) - Array changes frequently (after every few queries) - Array is small (< 1000 elements)

Algorithm 2: Dynamic Programming (Pre-calculate Everything!)

The Idea

What if we pre-calculate the answer for EVERY possible range? Then queries become instant lookups!

Building the DP Table Step-by-Step

Array: [5, 2, 4, 7]

We'll build a table where `dp[i][j]` = minimum value from index i to j.

Step 1: Single elements (length = 1)

```
dp[0][0] = 5  (just element at index 0)
dp[1][1] = 2  (just element at index 1)
dp[2][2] = 4  (just element at index 2)
dp[3][3] = 7  (just element at index 3)
```

Step 2: Pairs (length = 2)

```
dp[0][1] = min(5, 2) = 2
dp[1][2] = min(2, 4) = 2
dp[2][3] = min(4, 7) = 4
```

Step 3: Triples (length = 3)

```
dp[0][2] = min(dp[0][1], 4) = min(2, 4) = 2
dp[1][3] = min(dp[1][2], 7) = min(2, 7) = 2
```

Step 4: Full array (length = 4)

```
dp[0][3] = min(dp[0][2], 7) = min(2, 7) = 2
```

Final DP Table

```
    j→  0  1  2  3
i↓     ----------
0  |   5  2  2  2
1  |   -  2  2  2
2  |   -  -  4  4
3  |   -  -  -  7
```

Query Example

Query(1, 3): Just look up `dp[1][3] = 2`. Instant!

Time Complexity Deep Dive

Why is Preprocessing O(n²)? Let's count exactly how many cells we fill:

```
For array of size n:
- Ranges of length 1: n cells
- Ranges of length 2: n-1 cells
- Ranges of length 3: n-2 cells
- ...
- Ranges of length n: 1 cell

Total cells = n + (n-1) + (n-2) + ... + 1 = n(n+1)/2
```

Mathematical proof:

```
Sum = n(n+1)/2 = (n² + n)/2 = O(n²)
```

Actual operations per cell:

```
dp[i][j] = min(dp[i][j-1], array[j])
           ↑ 1 lookup + 1 comparison + 1 assignment = 3 operations
Total operations = 3 × n²/2 = O(n²)
```

Why is Query Time O(1)?

```
Query(L, R):
1. Access dp[L][R]  → 1 array access
2. Return value     → 1 operation
Total: 2 operations = O(1)
```

No loops, no comparisons, just direct memory access!

Space Complexity: O(n²)

```
For array of size n:
- DP table: n × n × 4 bytes = 4n² bytes
- Original array: n × 4 bytes = 4n bytes
Total: 4n² + 4n = O(n²)
```

Memory usage examples: - n = 100: ~40 KB - n = 1,000: ~4 MB - n = 10,000: ~400 MB (getting expensive!) - n = 100,000: ~40 GB (impractical!)

Building Time Analysis For an array of size n:

```
for (length = 1 to n):           → n iterations
    for (start = 0 to n-length): → average n/2 iterations
        dp[start][end] = ...      → O(1) operation
Total: n × n/2 × 1 = O(n²)
```

Real-world Performance For n = 1,000: - Preprocessing: ~2-3 milliseconds - Memory used: ~4 MB - Query time: ~0.01 microseconds (10 nanoseconds!) - Break-even point: Need ~2,000 queries to justify preprocessing

When to Use: - Array size < 2,000 elements - Number of queries > $n^2/1000$ - Can afford $O(n^2)$ memory - Static data (no updates)

Algorithm 3: Sparse Table (Binary Lifting Magic!)

The Clever Idea

Instead of storing ALL ranges, only store ranges with lengths that are powers of 2 (1, 2, 4, 8, ...). Any range can be covered by at most 2 overlapping power-of-2 ranges!

Building the Sparse Table Step-by-Step

Array: [5, 2, 4, 7, 1, 3, 6, 8]

We build `st[i][j]` = minimum in range starting at i with length 2^j.

Step 1: Length 1 (2^0 = 1)

```
st[0][0] = 5  (range [0,0])
st[1][0] = 2  (range [1,1])
st[2][0] = 4  (range [2,2])
st[3][0] = 7  (range [3,3])
st[4][0] = 1  (range [4,4])
st[5][0] = 3  (range [5,5])
st[6][0] = 6  (range [6,6])
st[7][0] = 8  (range [7,7])
```

Step 2: Length 2 (2^1 = 2)

```
st[0][1] = min(st[0][0], st[1][0]) = min(5, 2) = 2  (range [0,1])
st[1][1] = min(st[1][0], st[2][0]) = min(2, 4) = 2  (range [1,2])
st[2][1] = min(st[2][0], st[3][0]) = min(4, 7) = 4  (range [2,3])
st[3][1] = min(st[3][0], st[4][0]) = min(7, 1) = 1  (range [3,4])
st[4][1] = min(st[4][0], st[5][0]) = min(1, 3) = 1  (range [4,5])
st[5][1] = min(st[5][0], st[6][0]) = min(3, 6) = 3  (range [5,6])
st[6][1] = min(st[6][0], st[7][0]) = min(6, 8) = 6  (range [6,7])
```

Step 3: Length 4 (2^2 = 4)

```
st[0][2] = min(st[0][1], st[2][1]) = min(2, 4) = 2  (range [0,3])
st[1][2] = min(st[1][1], st[3][1]) = min(2, 1) = 1  (range [1,4])
st[2][2] = min(st[2][1], st[4][1]) = min(4, 1) = 1  (range [2,5])
st[3][2] = min(st[3][1], st[5][1]) = min(1, 3) = 1  (range [3,6])
st[4][2] = min(st[4][1], st[6][1]) = min(1, 6) = 1  (range [4,7])
```

Step 4: Length 8 (2^3 = 8)

```
st[0][3] = min(st[0][2], st[4][2]) = min(2, 1) = 1  (range [0,7])
```

Sparse Table Visualization

```
          Length→
Index↓    1   2   4   8
0         5   2   2   1
```

```
1        2   2   1   -
2        4   4   1   -
3        7   1   1   -
4        1   1   1   -
5        3   3   -   -
6        6   6   -   -
7        8   -   -   -
```

Query Example: Query(2, 6)

```
Range length = 6 - 2 + 1 = 5
Largest power of 2   5 is 4 (2^2)

Split into two overlapping ranges of length 4:
- Range 1: [2, 5] → st[2][2] = 1
- Range 2: [3, 6] → st[3][2] = 1

Answer: min(1, 1) = 1
```

Visual of Query Coverage

```
Query [2, 6]:
Index: 0  1  2  3  4  5  6  7
Value: 5  2  4  7  1  3  6  8
            |-----------|     (st[2][2]: covers 2-5)
               |-----------|  (st[3][2]: covers 3-6)
            |============|    (Full coverage with overlap!)
```

Time Complexity Deep Dive

Why is Preprocessing O(n log n)? **Understanding log n levels:**

```
For array of size n = 16:
- Level 0 (length 1):  16 entries
- Level 1 (length 2):  15 entries
- Level 2 (length 4):  13 entries
- Level 3 (length 8):  9 entries
- Level 4 (length 16): 1 entry

Number of levels = log (16) + 1 = 5 levels
```

Counting total operations:

```
for j from 0 to log(n):              → log n iterations
    for i from 0 to n - 2^j:         → (n - 2^j + 1) iterations
        st[i][j] = min(st[i][j-1], st[i+2^(j-1)][j-1])
                   ↑ 2 lookups + 1 comparison + 1 assignment

Total entries = n×1 + (n-1)×1 + (n-3)×1 + ...
                n × log n entries
```

```
Each entry: O(1) operation
Total: O(n log n)
```

Mathematical analysis:

```
Sum = Σ(j=0 to log n) of (n - 2^j + 1)
    = n×log(n) - (2^(log n+1) - 1) + log(n)
    = n×log(n) - (2n - 1) + log(n)
    = O(n log n)
```

Why is Query Time O(1)? **The brilliant trick:**

```
Query(L, R):
1. Calculate k = floor(log (R - L + 1))  → O(1) with bit operations
2. Access st[L][k]                        → O(1) array access
3. Access st[R - 2^k + 1][k]             → O(1) array access
4. Return min of the two                  → O(1) comparison
Total: 4 operations = O(1)
```

How to calculate log in O(1):

```
// Using built-in functions (compiled to single CPU instruction)
int k = __builtin_clz(1) - __builtin_clz(R - L + 1);
// or
int k = 31 - __builtin_clz(R - L + 1);  // for 32-bit integers
```

Space Complexity: O(n log n)

```
Sparse table dimensions:
- Rows: n (one for each starting position)
- Columns: log (n) + 1 (one for each power of 2)
- Each cell: 4 bytes (integer)

Total space = n × (log n + 1) × 4 bytes = O(n log n)
```

Memory usage examples: - n = 1,000: ~40 KB (log (1000)   10) - n = 100,000: ~6.4 MB (log (100000)   17) - n = 1,000,000: ~80 MB (log (1000000)   20)

Much better than DP's O(n^2)!

Why Can We Overlap Ranges? This only works because MIN is an **idempotent** operation:

```
min(a, a) = a
min(min(a,b), min(b,c)) = min(a,b,c)
```

So overlapping doesn't affect the result!

Real-world Performance For n = 100,000: - Preprocessing: ~15-20 milliseconds - Memory used: ~6.4 MB - Query time: ~0.05 microseconds (50 nanoseconds) - Can handle millions of queries per second!

Comparison with DP: - DP for n=100,000: 40 GB memory (impractical) - Sparse Table: 6.4 MB memory (very practical) - Both have O(1) query, but Sparse Table scales much better

When to Use: - Large static arrays (up to 10 elements) - Need absolutely fastest query time - Can afford O(n log n) preprocessing - No updates to the array

Algorithm 4: Block Decomposition (Square Root Decomposition)

The Balanced Idea

Divide the array into blocks of size √n. Pre-compute the minimum for each complete block. For queries, combine partial blocks with complete blocks.

Building Blocks Step-by-Step

Array: [5, 2, 4, 7, 1, 3, 6, 8, 9] (n = 9, block_size = 3)

Step 1: Divide into blocks

```
Block 0: [5, 2, 4]
Block 1: [7, 1, 3]
Block 2: [6, 8, 9]
```

Step 2: Pre-compute block minimums

```
block_min[0] = min(5, 2, 4) = 2
block_min[1] = min(7, 1, 3) = 1
block_min[2] = min(6, 8, 9) = 6
```

Visual Representation

```
Array:  [5, 2, 4] [7, 1, 3] [6, 8, 9]
         Block 0   Block 1   Block 2
Min:        2         1         6
```

Query Example: Query(1, 7)

Step 1: Identify affected blocks

```
Index 1 is in Block 0 (partial)
Index 7 is in Block 2 (partial)
Block 1 is completely covered
```

Step 2: Calculate minimum

```
Partial Block 0: elements [1, 2] → min(2, 4) = 2
Complete Block 1: block_min[1] = 1
Partial Block 2: element [6, 7] → min(6, 8) = 6

Answer: min(2, 1, 6) = 1
```

Visual of Query

```
Query [1, 7]:
[5, |2, 4] [7, 1, 3] [6, 8|, 9]
    ↑~~~~   ~~~~~~~~   ~~~~↑
    partial complete  partial
```

Time Complexity Deep Dive

Why is Preprocessing O(n)?

```
Preprocessing steps:
1. Determine block size = √n           → O(1)
2. Create block_min array of size √n   → O(√n)
3. For each element in array:          → n iterations
   - Assign to a block                 → O(1)
   - Update block minimum              → O(1)
Total: O(1) + O(√n) + n×O(1) = O(n)
```

Detailed breakdown:

```
block_size = sqrt(n);                      // O(1)
for (int i = 0; i < n; i++) {             // n iterations
    int block_id = i / block_size;         // O(1) - integer division
    block_min[block_id] = min(block_min[block_id], arr[i]); // O(1)
}
Total: O(n)
```

Why is Query Time O(√n)? **Three parts of any query:**

1. **Left partial block:** Up to √n - 1 elements
2. **Complete middle blocks:** Up to √n - 2 blocks (each takes O(1) to check)
3. **Right partial block:** Up to √n - 1 elements

```
Worst case analysis:
- Left partial: √n - 1 comparisons
- Middle blocks: √n - 2 lookups
- Right partial: √n - 1 comparisons
Total: (√n - 1) + (√n - 2) + (√n - 1) = 3√n - 4 = O(√n)
```

Example with n = 100 (block_size = 10):

```
Query(5, 84):
- Left partial [5-9]: 5 elements     → 5 operations
- Middle blocks [10-79]: 7 blocks    → 7 operations
- Right partial [80-84]: 5 elements  → 5 operations
Total: 17 operations   1.7√n
```

Why Block Size = √n is Optimal? Let's analyze with block size = b:

```
- Number of blocks: n/b
- Elements per block: b
- Query time: O(b) for partials + O(n/b) for complete blocks

Total query time: O(b + n/b)
To minimize, take derivative and set to 0:
d/db (b + n/b) = 1 - n/b² = 0
b² = n
b = √n
```

What if we use different block sizes? - Block size = n/10: Query = O(n/10), not good! - Block size = 10: Query = O(n/10), still linear! - Block size = $\sqrt{n}$: Query = $O(\sqrt{n})$, perfectly balanced!

Space Complexity: $O(\sqrt{n} + n)$

```
Storage requirements:
- Original array: n elements × 4 bytes = 4n bytes
- Block minimums: √n blocks × 4 bytes = 4√n bytes
- Block boundaries: 2√n integers = 8√n bytes (optional)
Total: O(n + √n) = O(n)
```

But we often say $O(\sqrt{n})$ for *additional* space beyond the input array.

Update Complexity: O(1)   This is where Block Decomposition shines!

```
Update(index, new_value):
1. Update array[index] = new_value     → O(1)
2. Find block_id = index / block_size  → O(1)
3. Recompute minimum for that block    → O(√n)
   OR
   If new_value < block_min: update   → O(1)
   If old_value was min: recompute    → O(√n)
```

Smart update strategy: - If new value is smaller than block min: O(1) - Otherwise: $O(\sqrt{n})$ to recompute one block - Compare to other algorithms: - Sparse Table: O(n log n) rebuild everything! - DP: $O(n^2)$ rebuild everything!

Real-world Performance   For n = 1,000,000 (block_size   1,000): - Preprocessing: ~1 millisecond - Memory used: ~4 KB extra (just block minimums) - Query time: ~1 microsecond - Update time: ~1 microsecond (average case)

Performance comparison:

Algorithm	Query	Update	Memory
Naive	1000 s	0.01 s	0
Block Decomp	1 s	1 s	4 KB
Sparse Table	0.05 s	20,000 s	80 MB

Mathematical Beauty: The Square Root Appears Everywhere!   For array of size n with block size $\sqrt{n}$: - Number of blocks: $\sqrt{n}$ - Elements per block: $\sqrt{n}$ - Query touches at most: $2\sqrt{n}$ elements + $\sqrt{n}$ blocks = $3\sqrt{n}$ - Update affects: $\sqrt{n}$ elements (one block) - Extra space: $\sqrt{n}$ block minimums

When to Use: - Need both queries and updates - Array size up to 10 elements - Can't afford O(n log n) space - Updates are as common as queries

Algorithm 5: LCA-based RMQ (The Tree Transformation!)

The Amazing Connection

RMQ can be transformed into finding the Lowest Common Ancestor (LCA) in a tree! We build a special tree called a Cartesian Tree.

Building the Cartesian Tree Step-by-Step

Array: [5, 2, 4, 7, 1, 3]

Rules for Cartesian Tree:

1. In-order traversal gives the original array
2. Parent is always smaller than children (min-heap property)

Construction Process: **Step 1:** Add 5

```
5
```

Step 2: Add 2 (smaller than 5, becomes new root)

```
2
 \
  5
```

Step 3: Add 4 (larger than 2, smaller than 5)

```
2
 \
  4
   \
    5
```

Step 4: Add 7 (larger than all)

```
2
 \
  4
   \
    5
     \
      7
```

Step 5: Add 1 (smallest, becomes new root)

```
    1
   / \
  2   3
   \
    4
     \
      5
       \
              7
```

Wait, that's not right! Let me rebuild properly:

Step 5: Add 1 (smallest so far, becomes new root)

```
    1
   /
  2
   \
    4
     \
      5
       \
        7
```

Step 6: Add 3 (larger than 1, goes to right)

```
      1
     / \
    2   3
     \
      4
       \
        5
         \
          7
```

Final Cartesian Tree

```
Array: [5, 2, 4, 7, 1, 3]
Index:  0  1  2  3  4  5

Tree:
        1(idx:4)
       /        \
    2(idx:1)   3(idx:5)
   /        \
5(idx:0)   4(idx:2)
              \
            7(idx:3)
```

How RMQ becomes LCA

Key Insight: The minimum element in range [L, R] is the LCA of nodes at positions L and R!

Example: Query(1, 3) - Find min between indices 1 and 3

1. Find nodes at indices 1 and 3 in the tree
    - Index 1 → Node with value 2
    - Index 3 → Node with value 7

2. Find their LCA:
    - Path from 2 to root: 2 → 1
    - Path from 7 to root: 7 → 4 → 2 → 1
    - Common ancestor: 2
3. Answer: 2

LCA using Binary Lifting

We store ancestors at powers of 2 distances:

Ancestor Table

Node	Parent	2^1-ancestor	2^2-ancestor
1	null	null	null
2	1	null	null
5	2	1	null
4	2	1	null
7	4	2	null
3	1	null	null

Time Complexity Deep Dive

Building the Cartesian Tree: O(n) **The amazing linear-time construction using a stack:**

```
stack<int> st;
for (int i = 0; i < n; i++) {           // n iterations
    int last_popped = -1;
    while (!st.empty() && arr[st.top()] > arr[i]) {  // amortized O(1)
        last_popped = st.top();
        st.pop();
    }
    // Set parent-child relationships - all O(1)
    if (!st.empty()) right_child[st.top()] = i;
    if (last_popped != -1) left_child[i] = last_popped;
    st.push(i);
}
```

Why is the while loop O(1) amortized? - Each element is pushed exactly once: n pushes total - Each element is popped at most once: n pops total - Total operations across all iterations: 2n = O(n) - Amortized per iteration: O(1)

LCA Preprocessing: O(n log n) for Binary Lifting **Building the ancestor table:**

```
For each node (n nodes):
    For each power of 2 up to log n:
        ancestor[node][j] = ancestor[ancestor[node][j-1]][j-1]

Total: n × log n entries × O(1) per entry = O(n log n)
```

Detailed breakdown:

```
Level 0: Store immediate parents        → n × 1 = n operations
Level 1: Store 2-hop ancestors         → n × 1 = n operations
Level 2: Store 4-hop ancestors         → n × 1 = n operations
...
Level log n: Store 2^(log n) ancestors → n × 1 = n operations

Total: n × (log n + 1) = O(n log n)
```

Query Time: O(log n) for Binary Lifting LCA Finding LCA of nodes u and v:

```
1. Bring u and v to same depth:
   - Calculate depths               → O(1) if preprocessed
   - Jump up using binary lifting   → O(log n) jumps maximum

2. Binary search for LCA:
   for j from log n down to 0:      → log n iterations
       if ancestor[u][j] != ancestor[v][j]:
           u = ancestor[u][j]
           v = ancestor[v][j]       → O(1) per iteration

   LCA = parent[u]                  → O(1)

Total: O(log n) + O(log n) = O(log n)
```

Example with depth difference = 13:

```
13 in binary = 1101
Jump by: 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13
Number of jumps: 3 (number of 1s in binary)
Maximum jumps: log n
```

Alternative: O(1) Query with Euler Tour + RMQ Preprocessing steps: 1. Build Cartesian Tree: O(n) 2. Euler Tour of tree: O(n) - visit each edge twice 3. Build RMQ on depths: O(n) for ±1 RMQ Total: O(n)

Query: 1. Find first occurrence of u and v in tour: O(1) with preprocessing 2. RMQ on depth array between them: O(1) with ±1 RMQ Total: O(1)

This gives us O(n) preprocessing and O(1) query!

Space Complexity Analysis Binary Lifting approach:

```
- Cartesian Tree: n nodes × 3 pointers = 3n pointers
- Ancestor table: n nodes × log n levels = n log n entries
- Depth array: n integers
Total: O(n log n)
```

Euler Tour approach:

```
- Cartesian Tree: 3n pointers
- Euler tour: 2n - 1 entries
- Depth array: 2n - 1 entries
- First occurrence: n entries
- ±1 RMQ structure: O(n)
Total: O(n)
```

Why Transform RMQ to LCA? **Theoretical importance:** - Shows RMQ LCA (equivalent problems) - Any LCA solution gives RMQ solution - Any RMQ solution gives LCA solution - Unifies two seemingly different problems

Practical benefits: - Reuse existing LCA code - Some LCA variants are easier to solve - Opens door to other tree algorithms

Real-world Performance For n = 100,000: - Cartesian tree construction: ~10 ms - LCA preprocessing: ~15 ms - Total preprocessing: ~25 ms - Query time: ~0.2 microseconds - Memory: ~8 MB

Comparison with direct approaches:

Algorithm	Preprocessing	Query	Theoretical Interest
Sparse Table	12 ms	0.05 s	Low
LCA-based	25 ms	0.20 s	Very High

The LCA approach is slightly slower but demonstrates beautiful theoretical connections!

When to Use: - You already have LCA code - Working with tree-related problems - Need to understand RMQ-LCA equivalence - Academic/competitive programming context

Comparison: Which Algorithm to Choose?

Quick Decision Guide

```
Need fast updates?
  YES → Use Naive (no preprocessing) or Block Decomposition
  NO → Continue...

   Need O(1) queries?
     YES → Use Sparse Table (best) or DP (if n < 1000)
     NO → Continue...

      Array size > 10000?
        YES → Use Sparse Table or Block Decomposition
        NO → Use DP (simplest for small arrays)
```

Performance Summary Table

Algorithm	Build Time	Query Time	Space	Best For
Naive	O(1)	O(n)	O(n)	Rare queries, frequent updates
DP	O(n^2)	O(1)	O(n^2)	Small arrays, many queries
Sparse Table	O(n log n)	O(1)	O(n log n)	Static arrays, fastest queries
Block Decomp	O(n)	O(√n)	O(√n)	Balanced operations
LCA-based	O(n log n)	O(log n)	O(n log n)	Theoretical interest

Interactive Examples

Let's Trace Through a Complete Example

Array: [3, 1, 4, 1, 5, 9, 2, 6]

Query(2, 6) = Find minimum in range [2, 6]   Naive Approach:

```
Check index 2: value = 4, min = 4
Check index 3: value = 1, min = 1
Check index 4: value = 5, min = 1
Check index 5: value = 9, min = 1
Check index 6: value = 2, min = 1
Answer: 1
```

DP Approach:

```
Look up dp[2][6] = 1 (pre-calculated)
Answer: 1
```

Sparse Table:

```
Range length = 5
Largest power of 2   5 = 4
Query ranges: [2,5] and [3,6]
st[2][2] = 1, st[3][2] = 1
Answer: min(1, 1) = 1
```

Block Decomposition (block_size = 3):

```
Blocks: [3,1,4] [1,5,9] [2,6,-]
Query spans partial block 0, complete block 1, partial block 2
Partial 0: min(4) = 4
Complete 1: block_min = 1
Partial 2: min(2) = 2
Answer: min(4, 1, 2) = 1
```

Practice Problems

Problem 1: Build Your Own DP Table

Array: [4, 2, 3, 1]

Fill in the DP table:

```
    j→  0  1  2  3
i↓     ----------
0  |   ?  ?  ?  ?
1  |   -  ?  ?  ?
2  |   -  -  ?  ?
3  |   -  -  -  ?
```

Solution

```
    j→  0  1  2  3
i↓     ----------
0  |   4  2  2  1
1  |   -  2  2  1
2  |   -  -  3  1
3  |   -  -  -  1
```

Problem 2: Sparse Table Query

Given sparse table for array [6, 2, 5, 1, 7, 3]:

```
st[0][0]=6, st[0][1]=2, st[0][2]=1
st[1][0]=2, st[1][1]=2, st[1][2]=1
st[2][0]=5, st[2][1]=1, st[2][2]=1
st[3][0]=1, st[3][1]=1
st[4][0]=7, st[4][1]=3
st[5][0]=3
```

What is Query(1, 4)?

Solution

Range length = 4, use k = 2 (2^2 = 4) Query ranges: [1,4] covered by st[1][2] = 1 Answer: 1

Tips and Tricks

1. Sparse Table Power-of-2 Trick

Use bit operations for fast power-of-2 calculations:

```
int k = __builtin_clz(1) - __builtin_clz(range_length);
// or
int k = floor(log2(range_length));
```

2. Block Size Selection

For Block Decomposition, optimal block size is usually: - √n for balanced operations - Smaller blocks for faster queries - Larger blocks for faster updates

3. DP Memory Optimization

Only need to store the upper triangle of the DP table since dp[i][j] only makes sense when i   j.

4. Cartesian Tree Stack Trick

Build Cartesian Tree in O(n) using a stack to maintain the right spine of the tree.

Deep Dive: Understanding Complexity Growth

How Complexities Compare As N Grows

Let’s see how each complexity grows with input size:

n	log n	√n	n	n log n	n^2
10	3	3	10	33	100
100	7	10	100	664	10,000
1,000	10	32	1,000	10,000	1,000,000
10,000	13	100	10,000	133,000	100,000,000
100,000	17	316	100,000	1,700,000	10,000,000,000
1,000,000	20	1,000	1,000,000	20,000,000	1,000,000,000,000

Time Complexity Visualization

```
Operations vs Input Size (log scale)

10^12 |                                               n²
10^10 |
10^8  |
10^6  |                             n log n
10^4  |                   n
10^2  |       √n
10^0  |   log n
      |________________________________
       10   100   1K   10K  100K  1M
```

When Each Algorithm Wins

Scenario 1: Few Queries (< 100)

```
Total Time = Preprocessing + (Number of Queries × Query Time)

For 50 queries on n = 100,000:
- Naive: 0 + 50 × 1000 s = 50,000 s   (Winner!)
- Sparse Table: 20,000 s + 50 × 0.05 s = 20,002 s
- Block: 100 s + 50 × 1 s = 150 s
```

Scenario 2: Many Queries (> 1,000,000)

```
For 1,000,000 queries on n = 100,000:
- Naive: 0 + 1M × 1000 s = 1,000,000,000 s (16 minutes!)
- Sparse Table: 20,000 s + 1M × 0.05 s = 70,000 s   (Winner!)
- Block: 100 s + 1M × 1 s = 1,000,100 s
```

Scenario 3: Queries with Updates

```
For 10,000 queries + 1,000 updates on n = 100,000:
- Naive: 10,000 × 1000 s + 1,000 × 0 = 10,000,000 s
- Block: 10,000 × 1 s + 1,000 × 1 s = 11,000 s   (Winner!)
- Sparse Table: Must rebuild after each update = Terrible!
```

Memory vs Speed Trade-offs

Algorithm	Memory	Query Speed	Can Update?
Naive	4MB	Slow	Instant
DP	40GB	Fastest	Rebuild all
Sparse Table	6.4MB	Fastest	Rebuild all
Block	4MB	Fast	Fast
LCA	8MB	Fast	Rebuild tree

Big-O Doesn't Tell the Whole Story!

Hidden Constants Matter Two O(n) algorithms can differ by 100x in practice:

```c
// Algorithm A: O(n) with small constant
for (int i = 0; i < n; i++)
    sum += arr[i];  // 1 operation per iteration

// Algorithm B: O(n) with large constant
for (int i = 0; i < n; i++) {
    result = complex_hash(arr[i]);     // 50 operations
    result = expensive_check(result);  // 30 operations
    sum += result;                      // 80 operations total
}
```

Both are O(n), but B is 80x slower!

Cache Performance Modern CPUs have cache hierarchies: - L1 Cache: 0.5 ns access time - L2 Cache: 7 ns access time - Main Memory: 100 ns access time

```
Sequential access (cache-friendly):
arr[0], arr[1], arr[2], ...  → All from cache!

Random access (cache-unfriendly):
arr[1000], arr[0], arr[5000], ...  → Cache misses!
```

This is why Sparse Table (sequential access) often beats theoretically faster algorithms with random access patterns.

Amortized Analysis: When Average Case Matters

Example: Dynamic Array Resizing

```
Push operations: 1, 1, 1, (resize+copy:4), 1, 1, 1, 1, (resize+copy:8), ...

Individual operations: O(1) usually, O(n) sometimes
Amortized (average): O(1) per operation!
```

In RMQ Context Block Decomposition updates: - Best case: New min is smaller → O(1) - Worst case: Recompute block → O(√n) - Amortized: Often O(1) in practice

The Complexity Hierarchy

```
O(1)  O(log n)  O(√n)  O(n)  O(n log n)  O(n²)  O(2^n)
```

```
Constant < Logarithmic < Sublinear < Linear < Linearithmic < Quadratic < Exponential
```

Rule of thumb for max input sizes: - O(1), O(log n): Any size (limited by memory) - O(√n): Up to 10^14 - O(n): Up to 10^8 - O(n log n): Up to 10^6 - O(n^2): Up to 10^4 - O(n^3): Up to 500 - O(2^n): Up to 20

Conclusion

Each RMQ algorithm represents a different trade-off between preprocessing time, query time, space usage, and update capability. Understanding not just the big-O notation but also:

- Hidden constants in the implementation
- Cache performance characteristics
- Amortized vs worst-case behavior
- Memory access patterns
- Practical input size limits

…helps you choose the right algorithm for your specific use case.

Remember: - **Naive** = No prep, just scan (best for rare queries) - **DP** = Pre-calculate everything (impractical for large arrays) - **Sparse Table** = Smart power-of-2 ranges (best for static arrays) - **Blocks** = Divide and conquer (best with updates) - **LCA** = Transform to tree problem (theoretical elegance)

The "best" algorithm depends entirely on your specific requirements!

Happy querying!