Segment Trees

Data Structures

Minseok Jeon
DGIST

November 2, 2025

Table of Contents

Introduction to Segment Trees

What is a Segment Tree?

Segment Tree: A binary tree structure for efficient range queries and updates

Key Properties:

- Each node represents an interval/segment of the array
- Leaf nodes represent single elements
- Internal nodes store aggregate information about their range
- Height: O(log n) for n elements

Common Operations:

- Range sum/min/max queries
- Point updates
- Range updates (with lazy propagation)

Time Complexity: O(log n) for queries and updates

Motivation

Problem: Given array, answer multiple queries:

- What is the sum of elements from index i to j?
- Update element at index k

Naive Approach:

- Query: O(n) - iterate through range
- Update: O(1) - change single element

Prefix Sum Approach:

- Query: O(1) - prefix[j] - prefix[i-1]
- Update: O(n) - rebuild prefix array

Segment Tree Solution:

- Query: O(log n) - traverse tree
- Update: O(log n) - update path to root
- Best of both worlds!

Applications

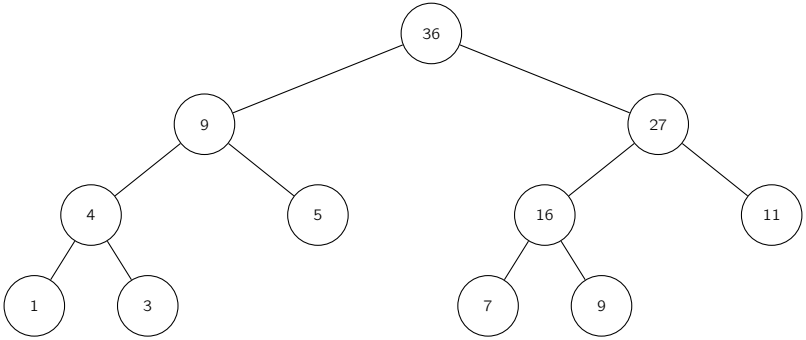
Segment trees are versatile and widely used:

1. **Competitive Programming:**
   - Range sum/min/max queries with updates
   - Finding k-th smallest element in range
   - Maximum subarray sum in range
2. **Databases:**
   - Range aggregation queries
   - Time-series databases
   - Spatial indexes
3. **Graphics & Games:**
   - Rectangle union area
   - Collision detection
   - Visibility queries
4. **Other:**
   - Network monitoring
   - Financial OHLC queries
   - Computational geometry

Building Segment Trees

Tree Structure

Array: [1, 3, 5, 7, 9, 11]



Each node stores: **sum** of its range

Array Representation

Tree stored in array: Similar to heap representation

Index mapping:

- Node at index i has children at $2i$ and $2i + 1$
- Parent of node i is at $i/2$
- Root at index 1

Space requirement:

- For array of size n, need $\approx 4n$ space
- Height = $\lceil \log_2 n \rceil$
- Total nodes = $2^{h+1} - 1 < 4n$

Array layout for [1, 3, 5, 7, 9, 11]:

Index	1	2	3	4	5	6	7	8	9	12	13
Value	36	9	27	4	5	16	11	1	3	7	9

Building the Tree - Recursive

```python
class SegmentTree:
    def __init__(self, arr):
        self.n = len(arr)
        self.tree = [0] * (4 * self.n)
        self.arr = arr
        if self.n > 0:
            self.build(1, 0, self.n - 1)

    def build(self, node, start, end):
        """
        Build segment tree recursively
        node: current node index in tree array
        [start, end]: range this node represents
        """
        if start == end:
            # Leaf node - copy array element
            self.tree[node] = self.arr[start]
            return
```

Building the Tree - Iterative

More efficient approach using 2n space:

```python
def build_iterative(arr):
    """Build segment tree iteratively - more efficient"""
    n = len(arr)
    tree = [0] * (2 * n)

    # Copy array to second half
    for i in range(n):
        tree[n + i] = arr[i]

    # Build tree by calculating parents
    for i in range(n - 1, 0, -1):
        tree[i] = tree[2 * i] + tree[2 * i + 1]

    return tree
```

Advantages:

Range Queries

Query Types

Segment trees support various aggregate queries:

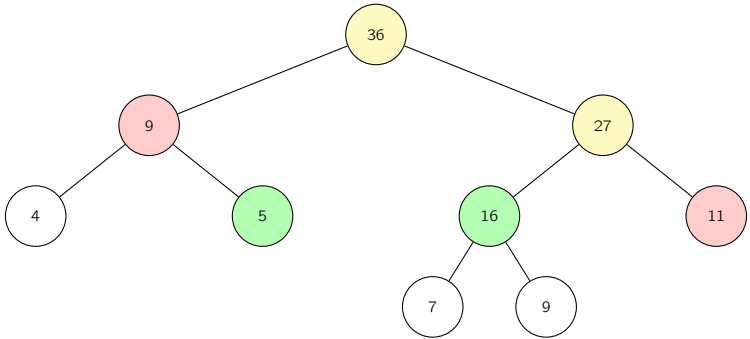
1. **Range Sum:**
   - Query: sum(left, right) = sum of elements in [left, right]
   - Combine: $\text{sum}_{\text{left}} + \text{sum}_{\text{right}}$
2. **Range Minimum:**
   - Query: min(left, right) = minimum element in [left, right]
   - Combine: $\min(\text{min}_{\text{left}}, \text{min}_{\text{right}})$
3. **Range Maximum:**
   - Query: max(left, right) = maximum element in [left, right]
   - Combine: $\max(\text{max}_{\text{left}}, \text{max}_{\text{right}})$
4. **Other Operations:**
   - GCD of range
   - XOR of range

Range Sum Query Implementation

```python
def query_sum(self, node, start, end, left, right):
    """
    Query sum in range [left, right]
    node: current node
    [start, end]: range of current node
    [left, right]: query range
    """
    # Case 1: No overlap
    if right < start or left > end:
        return 0

    # Case 2: Complete overlap
    if left <= start and end <= right:
        return self.tree[node]

    # Case 3: Partial overlap
    mid = (start + end) // 2
    left_sum = self.query_sum(2 * node, start, mid, left, right)
    right_sum = self.query_sum(2 * node + 1, mid + 1, end,
```

Query Visualization

Query: sum(2, 4) in array [1, 3, 5, 7, 9, 11]



Green: Complete overlap Yellow: Partial overlap Red: No overlap

Result: 5 + 16 = 21 (which equals 5 + 7 + 9 ✓)

Range Minimum Query

Similar structure, different combine operation:

```python
class SegmentTreeMin:
    def __init__(self, arr):
        self.n = len(arr)
        self.tree = [float('inf')] * (4 * self.n)
        self.arr = arr
        if self.n > 0:
            self.build(1, 0, self.n - 1)

    def build(self, node, start, end):
        if start == end:
            self.tree[node] = self.arr[start]
            return

        mid = (start + end) // 2
        self.build(2 * node, start, mid)
        self.build(2 * node + 1, mid + 1, end)

        # Store minimum of children
```

Query Time Complexity

Why is query O(log n)?

Analysis:

- Tree height: $h = \lceil \log_2 n \rceil$
- At each level, visit at most 2 nodes
- Total nodes visited: $\leq 2h = O(\log n)$

Why at most 2 nodes per level?

1. Start at root
2. At each level, query range can intersect at most 2 children
3. Left boundary intersects one child
4. Right boundary intersects another child
5. Middle parts are completely covered (return immediately)

Best case: O(1) - complete overlap at root

Worst case: O(log n) - query single element

Point Updates

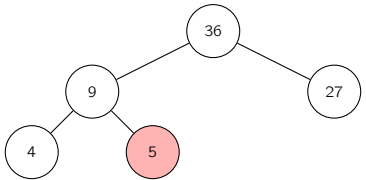
Point Update Implementation

```python
def update_point(self, node, start, end, idx, value):
    """
    Update element at index idx to value
    """
    if start == end:
        # Leaf node - update value
        self.arr[idx] = value
        self.tree[node] = value
        return

    mid = (start + end) // 2

    if idx <= mid:
        # Update in left subtree
        self.update_point(2 * node, start, mid, idx, value)
    else:
        # Update in right subtree
        self.update_point(2 * node + 1, mid + 1, end, idx, value)
```

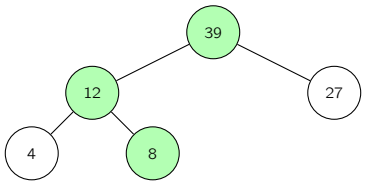
Update Visualization

Update index 2 from 5 to 8 (delta = +3)

Before:



After:



Update Time Complexity

Why is update O(log n)?

Path from leaf to root:

- Update travels from leaf to root
- Path length = tree height = $\lceil \log_2 n \rceil$
- Each ancestor needs updating

Example for n = 8:

- Height = 3
- Update path: leaf $\rightarrow$ parent $\rightarrow$ grandparent $\rightarrow$ root
- 4 nodes total = $\log_2 8 + 1$

Comparison:

- Naive array: O(1) update, but O(n) query
- Prefix sum: O(n) update (rebuild), O(1) query
- Segment tree: O(log n) update, O(log n) query ✓

Range Updates with Lazy Propagation

The Range Update Problem

Problem: Add delta to all elements in range [left, right]

Naive approach:
Time: O(n log n) - Too slow!

Problem:

- Range size can be O(n)
- Each update is O(log n)
- Total: O(n log n)

Solution: Lazy Propagation

- Mark nodes as "lazy" instead of updating immediately
- Defer updates until actually needed
- Push lazy values down only when queried
- Achieves O(log n) range update!

Lazy Propagation Concept

Key Idea: Don't update eagerly, be lazy!

Lazy Array:

- Additional array: `lazy[]` same size as tree
- `lazy[node]` = pending update for this node's range
- When `lazy[node]` $\neq$ `0`: node needs update

Update Strategy:

1. Find nodes completely covered by update range
2. Mark them as lazy (don't go deeper!)
3. Push lazy values down only when needed:
    - During next query
    - During next update that overlaps

Benefits:

- Range update: O(log n) instead of O(n log n)

Push Down Operation

```python
def push_down(self, node, start, end):
    """Propagate lazy value to children"""
    if self.lazy[node] != 0:
        # Apply lazy value to current node
        range_size = end - start + 1
        self.tree[node] += range_size * self.lazy[node]

        # If not leaf, propagate to children
        if start != end:
            self.lazy[2 * node] += self.lazy[node]
            self.lazy[2 * node + 1] += self.lazy[node]

        # Clear lazy value
        self.lazy[node] = 0
```

Key Points:

- Apply lazy value to current node
- Pass lazy value to children (don't apply recursively!)

Range Update with Lazy

```python
def update_range(self, node, start, end, left, right, delta):
    """Add delta to range [left, right]"""
    # Push down pending updates first
    self.push_down(node, start, end)

    # No overlap
    if right < start or left > end:
        return

    # Complete overlap
    if left <= start and end <= right:
        # Mark as lazy instead of updating
        self.lazy[node] += delta
        self.push_down(node, start, end)
        return

    # Partial overlap - recurse
    mid = (start + end) // 2
    self.update_range(2 * node, start, mid, left, right, delta)
    self.update_range(2 * node + 1, mid + 1, end, left, right, delta)

    # Update current node after children are updated
    self.push_down(2 * node, start, mid)
    self.push_down(2 * node + 1, mid + 1, end)
    self.tree[node] = self.tree[2 * node] + self.tree[2 * node + 1]

# Wrapper
def update(self, left, right, delta):
    self.update_range(1, 0, self.n - 1, left, right, delta)
```

Query with Lazy

```python
def query_range(self, node, start, end, left, right):
    """Query sum in range [left, right]"""
    # Push down pending updates first!
    self.push_down(node, start, end)

    # No overlap
    if right < start or left > end:
        return 0

    # Complete overlap
    if left <= start and end <= right:
        return self.tree[node]

    # Partial overlap
    mid = (start + end) // 2
    left_sum = self.query_range(2 * node, start, mid, left, right)
    right_sum = self.query_range(2 * node + 1, mid + 1, end,
                                 left, right)
```

Lazy Propagation Example

Array: [1, 2, 3, 4, 5]
Update: Add 10 to range [1, 3]

Without Lazy (Naive):

- Update index 1: O(log n)
- Update index 2: O(log n)
- Update index 3: O(log n)
- Total: O(3 log n) = O(n log n) for large ranges

With Lazy Propagation:

- Find O(log n) nodes covering [1, 3]
- Mark them as lazy: `lazy[node] = 10`
- Children not updated yet!
- Total: O(log n)

On Next Query:

Complexity Analysis

Space Complexity

Recursive Representation (4n):

- For n elements
- Tree height: $h = \lceil \log_2 n \rceil$
- Total nodes: $2^{h+1} - 1 \approx 4n$
- Safe upper bound: 4n

Iterative Representation (2n):

- More space-efficient
- Exactly 2n space
- First n positions: internal nodes
- Last n positions: leaf nodes (original array)

With Lazy Propagation:

- Additional lazy array: same size as tree
- Total: 8n (recursive) or 4n (iterative)

Time Complexity Summary

Operation	Without Lazy	With Lazy
Build	O(n)	O(n)
Point Update	O(log n)	O(log n)
Point Query	O(log n)	O(log n)
Range Query	O(log n)	O(log n)
Range Update	O(n log n)	O(log n)

Key Insight: Lazy propagation transforms range update from O(n log n) to O(log n)!

When to use lazy propagation:

- Frequent range updates
- Updates much more common than queries
- Batch updates before querying
- Willing to trade code complexity for performance

Comparison with Alternatives

Structure	Build	Query	Update	Space
Array	O(1)	O(n)	O(1)	O(n)
Prefix Sum	O(n)	O(1)	O(n)	O(n)
Sqrt Decomp	O(n)	$O(\sqrt{n})$	$O(\sqrt{n})$	O(n)
Segment Tree	O(n)	O(log n)	O(log n)	O(4n)
Fenwick Tree	O(n log n)	O(log n)	O(log n)	O(n)
Sparse Table	O(n log n)	O(1)	-	O(n log n)

When to use Segment Tree:

- Need range queries (sum, min, max, GCD, etc.)
- Need range updates
- Operation is associative
- Need flexibility (any associative operation)

Alternatives:

- **Fenwick Tree:** Less memory, only sum/XOR

Applications

Application 1: Range Sum Queries

Problem: Given array, answer queries and updates

```python
# LeetCode 307: Range Sum Query - Mutable
class NumArray:
    def __init__(self, nums):
        self.seg_tree = SegmentTree(nums)

    def update(self, index, val):
        self.seg_tree.update(index, val)

    def sumRange(self, left, right):
        return self.seg_tree.query(left, right)

# Usage
arr = [1, 3, 5, 7, 9, 11]
num_array = NumArray(arr)

print(num_array.sumRange(1, 4))  # 3+5+7+9 = 24
num_array.update(2, 10)          # Change 5 to 10
print(num_array.sumRange(1, 4))  # 3+10+7+9 = 29
```

Application 2: Range Minimum Query

Problem: Find minimum in range with updates

```
class RMQSegmentTree:
    def build(self, node, start, end):
        if start == end:
            self.tree[node] = self.arr[start]
            return

        mid = (start + end) // 2
        self.build(2 * node, start, mid)
        self.build(2 * node + 1, mid + 1, end)

        # Store minimum of children
        self.tree[node] = min(self.tree[2 * node],
                              self.tree[2 * node + 1])

    def query_min(self, node, start, end, left, right):
        if right < start or left > end:
            return float('inf')
```

Application 3: Count in Range

Problem: Count elements in range with value in [a, b]

Solution: Merge Sort Tree (segment tree of sorted arrays)

```
class MergeSortTree:
    def build(self, node, start, end):
        if start == end:
            self.tree[node] = [self.arr[start]]
            return

        mid = (start + end) // 2
        self.build(2 * node, start, mid)
        self.build(2 * node + 1, mid + 1, end)

        # Merge sorted arrays
        left_arr = self.tree[2 * node]
        right_arr = self.tree[2 * node + 1]
        self.tree[node] = sorted(left_arr + right_arr)

    def count_in_range(self, node, start, end, left, right, a, b):
        """Count elements in [left,right] with value in [a,b]"""
        if right < start or left > end:
            return 0

        if left <= start and end <= right:
            # Binary search in sorted array for count in [a, b]
            import bisect
            arr = self.tree[node]
            left_idx = bisect.bisect_left(arr, a)
            right_idx = bisect.bisect_right(arr, b)
            return right_idx - left_idx
```

Application 4: Database Range Aggregations

Time-Series Database:

- Store metrics over time
- Query: Get sum/avg/min/max over time range
- Segment tree enables O(log n) queries

Spatial Databases:

- 2D segment tree for spatial queries
- Query points in rectangle [x1, y1] to [x2, y2]
- Each node has nested segment tree for second dimension

Application 5: Computational Geometry

Rectangle Union Area:

- Given n rectangles, find total area covered
- Use sweep line + segment tree
- Track active intervals as sweep progresses

Closest Pair:

- Find closest pair of points
- Divide and conquer with segment tree
- Query minimum distance in ranges

Line Intersection:

- Count intersections among n line segments
- Sweep line algorithm
- Segment tree tracks active segments

Practice Problems

LeetCode Problems:

1. 307. Range Sum Query - Mutable
2. 327. Count of Range Sum
3. 699. Falling Squares
4. 715. Range Module
5. 732. My Calendar III
6. 850. Rectangle Area II

Codeforces/Other:

- SPOJ - GSS1, GSS3 (Maximum Subarray Sum)
- Codeforces - Range queries with various operations
- AtCoder - Lazy Segment Tree problems

Skills to Practice:

- Implementing different aggregation functions
- Lazy propagation

Summary

Summary: Key Concepts

Segment Tree Structure:

- Binary tree where each node represents an interval
- Leaf nodes: single elements
- Internal nodes: aggregate of children
- Height: O(log n), Space: O(4n)

Core Operations:

- Build: O(n)
- Range Query: O(log n)
- Point Update: O(log n)
- Range Update (with lazy): O(log n)

Lazy Propagation:

- Defer updates until needed
- Reduces range update from O(n log n) to O(log n)
- Essential for efficient range updates

Summary: When to Use

Use Segment Trees When:

- Need range queries (sum, min, max, GCD, etc.)
- Need frequent updates
- Operation is associative
- O(log n) performance required
- Need flexibility for different operations

Consider Alternatives When:

- **Fenwick Tree:** Only sum/XOR, want less memory
- **Sparse Table:** Static array, only queries
- **Sqrt Decomposition:** Simpler code, $O(\sqrt{n})$ OK
- **Array:** Few queries, frequent updates

Remember:

- Segment trees are versatile and powerful
- Trade-off: More memory for flexibility

Key Takeaways

1. Segment trees solve range query problems efficiently
2. O(log n) for both queries and updates
3. Lazy propagation is crucial for range updates
4. Works with any associative operation
5. Space: O(4n) - acceptable trade-off
6. Implementation: recursion or iteration
7. Applications: CP, databases, geometry, graphics
8. Practice is essential for mastery

Next Steps:

- Implement from scratch (without looking!)
- Solve LeetCode problems
- Try variations: 2D, dynamic, persistent
- Understand when NOT to use segment trees

Further Learning

Advanced Topics:

- Persistent Segment Trees (versioned)
- Dynamic Segment Trees (coordinate compression)
- 2D Segment Trees (nested structures)
- Segment Tree Beats (range chmin/chmax)
- Implicit Segment Trees (sparse)

Resources:

- CP-Algorithms: Segment Tree tutorial
- Codeforces: Segment tree blogs and problems
- YouTube: Tutorials on lazy propagation
- Competitive Programming books

Related Topics:

- Fenwick Tree (Binary Indexed Tree)
- Square Root Decomposition

Thank You!

Questions?

Segment Trees: Divide, Conquer, Query