Fenwick Trees

Binary Indexed Trees

Minseok Jeon
DGIST

November 2, 2025

Table of Contents

Introduction

What is a Fenwick Tree?

Fenwick Tree (Binary Indexed Tree / BIT): Compact array-based structure for efficient prefix sums and range queries

Key Features:

- Array-based: No pointers, cache-friendly
- Bit manipulation: Uses binary representation of indices
- Space-efficient: O(n) space
- Fast operations: O(log n) updates and queries
- Simple implementation: 20 lines of code

Core Operations:

- `update(i, delta)`: Add delta to element at index i
- `prefix_sum(i)`: Sum of elements from 1 to i
- `range_sum(l, r)`: Sum from l to r

All in O(log n) time!

Motivation

Problem: Given array, support:

1. Update: Change value at index i
2. Query: Find sum of elements from index l to r

Naive Solutions:

Approach	**Update**	**Query**
Simple Array	O(1)	O(n)
Prefix Sum Array	O(n)	O(1)
Fenwick Tree	**O(log n)**	**O(log n)**

Best of both worlds!

Applications

Fenwick Trees are widely used in:

Competitive Programming:

- Range sum queries with updates
- Counting inversions
- Order statistics (k-th smallest)

Database Systems:

- OLAP aggregate queries
- Time-series cumulative metrics
- Sliding window aggregations

Graphics & Games:

- 2D range sum queries (image processing)
- Particle system queries
- Collision detection

Binary Representation & LSB

The Key Insight: Binary Representation

Each index is responsible for a range based on its binary representation

Least Significant Bit (LSB):

- LSB(i) = rightmost set bit in binary representation of i
- Calculated using: `i & (-i)`
- Determines range size for index i

Examples:

- i = 12 (binary: 1100), LSB(12) = 4
- i = 10 (binary: 1010), LSB(10) = 2
- i = 7 (binary: 0111), LSB(7) = 1
- i = 8 (binary: 1000), LSB(8) = 8

Range covered by index i: [i - LSB(i) + 1, i]

LSB Calculation

Bit Manipulation Magic:

```python
def lsb(i):
    """Get least significant bit of i"""
    return i & (-i)

# How it works:
# i = 12  (binary: 0...01100)
# -i in two's complement:
#   Step 1: Invert bits  -> 1...10011
#   Step 2: Add 1        -> 1...10100
# i & (-i) = 0...00100 = 4
```

Why this works:

- Two's complement inverts all bits and adds 1
- All bits after LSB are flipped
- LSB itself stays in same position
- AND operation isolates the LSB

Responsibility Ranges

Each index stores cumulative sum for its range

Index	Binary	LSB	Range
1	0001	1	[1]
2	0010	2	[1-2]
3	0011	1	[3]
4	0100	4	[1-4]
5	0101	1	[5]
6	0110	2	[5-6]
7	0111	1	[7]
8	1000	8	[1-8]

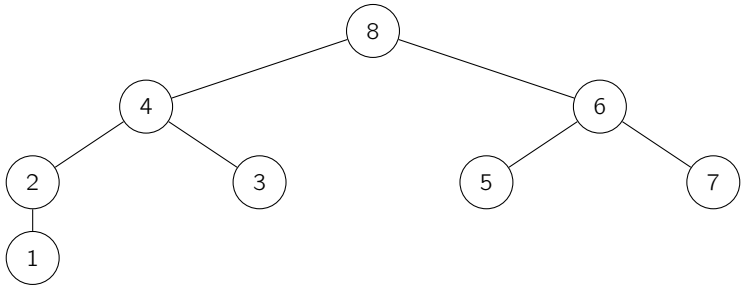
Pattern:

- Powers of 2 cover largest ranges
- Odd numbers cover single element
- Ranges never overlap in updates/queries

Tree Structure (Conceptual)

Although stored as array, conceptually forms a tree

For array of size 8:



Navigation:

- Parent: i + LSB(i) (move up)
- Previous range: i - LSB(i) (move left)

Core Operations

Point Update

Add delta to element at index i

```python
def update(self, i, delta):
    """Add delta to element at index i (1-indexed)"""
    while i <= self.n:
        self.tree[i] += delta
        i += i & (-i)  # Move to parent
```

How it works:

1. Start at index i
2. Update current node
3. Move to parent by adding LSB: i += LSB(i)
4. Repeat until out of bounds

Time Complexity: O(log n)

- At most $\log_2(n)$ iterations
- Each iteration: O(1) operations

Update Example

Update index 5 with delta = 3

Steps:

1. i = 5, LSB(5) = 1
   - `tree[5] += 3`
   - Next: 5 + 1 = 6
2. i = 6, LSB(6) = 2
   - `tree[6] += 3`
   - Next: 6 + 2 = 8
3. i = 8, LSB(8) = 8
   - `tree[8] += 3`
   - Next: 8 + 8 = 16 (stop if n < 16)

Result: Updated 3 nodes in O(log n) time!

These nodes represent all ranges containing index 5.

Prefix Sum Query

Get sum from index 1 to i

```python
def prefix_sum(self, i):
    """Get sum of elements from index 1 to i (1-indexed)"""
    result = 0
    while i > 0:
        result += self.tree[i]
        i -= i & (-i)  # Move to previous range
    return result
```

How it works:

1. Start at index i
2. Add current node's value to result
3. Move to previous non-overlapping range: i -= LSB(i)
4. Repeat until i = 0

Time Complexity: O(log n)

Prefix Sum Example

Query: prefix_sum(13)

Steps:

1. i = 13, LSB(13) = 1
   - `result += tree[13]` (covers [13])
   - Next: 13 - 1 = 12
2. i = 12, LSB(12) = 4
   - `result += tree[12]` (covers [9-12])
   - Next: 12 - 4 = 8
3. i = 8, LSB(8) = 8
   - `result += tree[8]` (covers [1-8])
   - Next: 8 - 8 = 0 (stop)

Ranges summed: [13] + [9-12] + [1-8] = [1-13] ✓

Range Sum Query

Get sum from index l to r

```
def range_sum(self, left, right):
    """Sum from left to right (1-indexed, inclusive)"""
    return self.prefix_sum(right) - self.prefix_sum(left - 1)
```

Key Insight:

- sum[l..r] = sum[1..r] - sum[1..l-1]
- Uses inclusion-exclusion principle
- Works because sum is invertible (subtraction defined)

Example:

```
# Query sum from index 3 to 7
sum[3..7] = prefix_sum(7) - prefix_sum(2)
```

Time Complexity: O(log n) - two prefix sum queries

Complete Implementation

```python
class FenwickTree:
    def __init__(self, n):
        self.n = n
        self.tree = [0] * (n + 1)  # 1-indexed

    def update(self, i, delta):
        """Add delta to index i (1-indexed)"""
        while i <= self.n:
            self.tree[i] += delta
            i += i & (-i)

    def prefix_sum(self, i):
        """Sum from index 1 to i (1-indexed)"""
        result = 0
        while i > 0:
            result += self.tree[i]
            i -= i & (-i)
        return result
```

Building the Tree

Initialization from Array

Two approaches to build tree from array:

Method 1: Using update (Simple but slower)

```
def __init__(self, arr):
    self.n = len(arr)
    self.tree = [0] * (self.n + 1)
    for i in range(self.n):
        self.update(i + 1, arr[i])
```

Time: O(n log n)

Method 2: Direct construction (Optimized)

```
def build_fast(arr):
    n = len(arr)
    tree = [0] * (n + 1)
    for i in range(1, n + 1):
        tree[i] += arr[i - 1]
        j = i + (i & -i)  # Parent index
```

Build Process Visualization

Direct O(n) construction:

For each index i:

1. Copy array value to tree[i]
2. Find parent: j = i + LSB(i)
3. Add tree[i] to tree[j]

Why this works:

- Process indices in increasing order
- When processing index i, all children already processed
- Parent inherits sum from child
- Each index processed exactly once: O(n)

Comparison:

- O(n log n): Easy to understand, uses existing update()
- O(n): More efficient, clever bottom-up approach

Advanced Techniques

Range Updates

Problem: Add delta to all elements in range [l, r]

Solution: Use difference array technique

```
class FenwickTreeRangeUpdate:
    def __init__(self, n):
        self.n = n
        self.tree = [0] * (n + 1)

    def range_update(self, left, right, delta):
        """Add delta to all elements in [left, right]"""
        self.update(left, delta)
        self.update(right + 1, -delta)

    def point_query(self, i):
        """Get value at index i after all updates"""
        return self.prefix_sum(i)
```

Key Idea:

Range Update Example

Add 5 to range [3, 6]

Initial array: [0, 0, 0, 0, 0, 0, 0, 0]

Operations:

1. `update(3, 5)` - Mark start of range
2. `update(7, -5)` - Mark end of range

Difference array: [0, 0, 5, 0, 0, 0, -5, 0]

Prefix sums (actual values):

- Index 1: 0
- Index 2: 0
- Index 3: 0 + 5 = 5
- Index 4: 5
- Index 5: 5
- Index 6: 5

2D Fenwick Tree

For 2D range sum queries

```python
class FenwickTree2D:
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.tree = [[0] * (cols + 1) for _ in range(rows + 1)]

    def update(self, row, col, delta):
        """Add delta to cell (row, col)"""
        i = row
        while i <= self.rows:
            j = col
            while j <= self.cols:
                self.tree[i][j] += delta
                j += j & (-j)
            i += i & (-i)

    def prefix_sum(self, row, col):
        """Sum of rectangle from (1,1) to (row, col)"""
        result = 0
        i = row
        while i > 0:
            j = col
            while j > 0:
                result += self.tree[i][j]
                j -= j & (-j)
            i -= i & (-i)
        return result

    def range_sum(self, r1, c1, r2, c2):
```

2D Range Sum

Query rectangle sum using inclusion-exclusion

Sum of rectangle (r1, c1) to (r2, c2):

```
sum = A - B - C + D
```

Where:

- A = prefix_sum(r2, c2) - full rectangle to (r2, c2)
- B = prefix_sum(r1-1, c2) - top portion (subtract)
- C = prefix_sum(r2, c1-1) - left portion (subtract)
- D = prefix_sum(r1-1, c1-1) - overlap (add back)

Complexity:

- Update: O(log m $\times$ log n)
- Query: O(log m $\times$ log n)
- Space: O(m $\times$ n)

Comparison with Segment Trees

Fenwick vs Segment Tree

Feature	Fenwick Tree	Segment Tree
Space	O(n)	O(4n)
Update	O(log n)	O(log n)
Query	O(log n)	O(log n)
Construction	O(n) or O(n log n)	O(n)
Code Lines	20	50
Operations	Sum, XOR	Any associative
Range Update	Difference array	Lazy propagation
Simplicity	Simple	Moderate
Constants	Lower	Higher

Key Difference: Fenwick Tree requires invertible operations!

When to Use Each

Use Fenwick Tree when:

- Operation is invertible (sum, XOR)
- Want minimal memory usage
- Prefer simple implementation
- Competitive programming (faster to code)
- Range sum queries with updates

Use Segment Tree when:

- Need range min/max queries
- Operation not invertible (min, max, GCD)
- Need lazy propagation
- Need to store extra data per node
- More complex range operations

Example where Fenwick fails:

- Range minimum: $\min(a, b) - \min(c, d)$ is meaningless!

Performance Comparison

Benchmark: 1,000,000 elements

Metric	Fenwick Tree	Segment Tree
Memory	4 MB	16 MB
Build time	20 ms	15 ms
1M updates	50 ms	80 ms
1M queries	50 ms	80 ms
Code size	20 lines	50 lines

Advantages of Fenwick:

- 4x less memory
- 1.6x faster for updates/queries
- 2.5x less code
- Better cache performance

Common Patterns & Pitfalls

Pattern: Counting Inversions

Problem: Count pairs (i, j) where i < j but arr[i] > arr[j]

```python
def count_inversions(arr):
    # Coordinate compression
    sorted_arr = sorted(set(arr))
    rank = {v: i+1 for i, v in enumerate(sorted_arr)}

    bit = FenwickTree(len(sorted_arr))
    inversions = 0

    # Process from right to left
    for i in range(len(arr) - 1, -1, -1):
        r = rank[arr[i]]
        # Count elements smaller than arr[i] seen so far
        inversions += bit.prefix_sum(r - 1)
        # Mark current element as seen
        bit.update(r, 1)

    return inversions
```

Time Complexity: O(n log n)

Applications:

- Sorting analysis
- Ranking systems

Common Pitfalls

1. **Index Confusion (0 vs 1-indexed)**
   - Fenwick Tree is 1-indexed!
   - Index 0 unused or sentinel
   - Always convert: `update(i + 1, delta)`

2. **Using for Non-Invertible Operations**
   - Cannot do range min/max with Fenwick
   - Subtraction doesn't work: min(a,b) - min(c,d)
   - Use Segment Tree instead

3. **Incorrect Size**
   - Array size must be n+1 (index 0 to n)
   - Check bounds in update loop

4. **Forgetting Negative Values**
   - Fenwick requires positive indices
   - Use coordinate compression for negative values

Pitfall Examples

Wrong: 0-indexed usage

```
bit = FenwickTree(n)
bit.update(0, 5)  # ERROR! Index 0 is invalid
```

Correct: Convert to 1-indexed

```
bit = FenwickTree(n)
bit.update(1, 5)  # OK: Update first element
```

Wrong: Range minimum

```
# Cannot use Fenwick for this!
min_val = bit.range_sum(l, r)  # Meaningless for min
```

Correct: Use Segment Tree

```
seg_tree = SegmentTree(arr)
min_val = seg_tree.range_min(l, r)  # OK
```

Debugging Tips

How to debug Fenwick Tree issues:

1. **Print tree array**
   - Visualize internal state
   - Check if values make sense
2. **Verify LSB calculation**
   - Test: `i & (-i)`
   - Check against manual calculation
3. **Check index conversions**
   - 0-indexed input $\rightarrow$ 1-indexed tree
   - Off-by-one errors common
4. **Test with small arrays**
   - Use n = 4 or 8 for manual verification
   - Compare with brute force
5. **Validate bounds**
   - Ensure i $\leq$ n in update
   - Ensure i > 0 in prefix_sum

Applications

Application 1: Range Sum Query

LeetCode 307: Range Sum Query - Mutable

```python
class NumArray:
    def __init__(self, nums):
        self.nums = nums
        self.n = len(nums)
        self.tree = [0] * (self.n + 1)
        for i in range(self.n):
            self._update(i + 1, nums[i])

    def update(self, index, val):
        delta = val - self.nums[index]
        self.nums[index] = val
        self._update(index + 1, delta)

    def _update(self, i, delta):
        while i <= self.n:
            self.tree[i] += delta
            i += i & (-i)
```

Application 2: Count Smaller Numbers

LeetCode 315: Count of Smaller Numbers After Self

```python
def countSmaller(nums):
    # Coordinate compression
    sorted_nums = sorted(set(nums))
    rank = {v: i+1 for i, v in enumerate(sorted_nums)}

    bit = FenwickTree(len(sorted_nums))
    result = []

    # Process from right to left
    for num in reversed(nums):
        r = rank[num]
        # Count numbers smaller than current
        count = bit.prefix_sum(r - 1)
        result.append(count)
        # Mark current number as seen
        bit.update(r, 1)

    return result[::-1]

class FenwickTree:
    def __init__(self, n):
        self.n = n
        self.tree = [0] * (n + 1)

    def update(self, i, delta):
        while i <= self.n:
            self.tree[i] += delta
            i += i & (-i)
```

Application 3: 2D Matrix Range Sum

LeetCode 308: Range Sum Query 2D - Mutable

```python
class NumMatrix:
    def __init__(self, matrix):
        if not matrix or not matrix[0]:
            return
        self.matrix = matrix
        self.rows = len(matrix)
        self.cols = len(matrix[0])
        self.tree = [[0] * (self.cols + 1) for _ in range(self.rows + 1)]

        for i in range(self.rows):
            for j in range(self.cols):
                self._update(i + 1, j + 1, matrix[i][j])

    def update(self, row, col, val):
        delta = val - self.matrix[row][col]
        self.matrix[row][col] = val
        self._update(row + 1, col + 1, delta)

    def _update(self, row, col, delta):
        i = row
        while i <= self.rows:
            j = col
            while j <= self.cols:
                self.tree[i][j] += delta
                j += j & (-j)
            i += i & (-i)

    def sumRegion(self, r1, c1, r2, c2):
        return (self._prefix_sum(r2+1, c2+1) - self._prefix_sum(r1, c2+1)
```

Real-World Usage

Competitive Programming:

- Codeforces: Preferred for speed
- AtCoder: Common in range query problems
- ICPC: Quick to implement under time pressure

Production Systems:

- Analytics databases (OLAP)
- Time-series aggregation
- Real-time monitoring dashboards
- Financial tick data analysis

Advanced Applications:

- Persistent Fenwick Trees (versioned queries)
- Parallel updates (lock-free concurrent operations)
- Compressed Fenwick Trees (sparse arrays)
- Multi-dimensional (3D, 4D spatial queries)

Summary

Summary: Key Concepts

Fenwick Tree Fundamentals:

- Array-based structure using binary representation
- LSB determines responsibility ranges
- Navigation via bit manipulation: $i \pm \text{LSB}(i)$

Core Operations:

- Update: O(log n) - add delta, move to parents
- Prefix sum: O(log n) - accumulate, move to previous ranges
- Range sum: O(log n) - difference of prefix sums

Key Properties:

- Space: O(n) - minimal overhead
- Simple: 20 lines of code
- Fast: Lower constants than Segment Tree
- Limitation: Requires invertible operations

Summary: Comparison

Structure	Space	Ops	Operations Supported
Array	O(n)	O(n)/O(1)	All
Prefix Sum	O(n)	O(n)/O(1)	Invertible
Sqrt Decomp	O(n)	$O(\sqrt{n})$	All
Fenwick	**O(n)**	**O(log n)**	**Invertible**
Segment Tree	O(4n)	O(log n)	All associative
Sparse Table	O(n log n)	O(log n)/O(1)	Idempotent

Sweet Spot:

- Best for range sum queries
- Optimal space-time trade-off
- Simplest code among log(n) structures

Key Takeaways

1. Fenwick Trees are elegant and efficient
2. Bit manipulation enables O(log n) operations
3. 1-indexed convention simplifies LSB logic
4. Perfect for invertible operations (sum, XOR)
5. Cannot handle min/max - use Segment Tree
6. Simpler and faster than Segment Tree when applicable
7. Essential tool for competitive programming
8. Useful in production for analytics systems

Remember:

- Always check if operation is invertible
- Master bit manipulation: `i & (-i)`
- Watch out for 0-indexed vs 1-indexed
- O(n) construction is possible and recommended

Practice Problems

LeetCode:

- 307. Range Sum Query - Mutable
- 308. Range Sum Query 2D - Mutable
- 315. Count of Smaller Numbers After Self
- 327. Count of Range Sum
- 493. Reverse Pairs

Codeforces:

- Fenwick tree problems (rated 1400-2000)
- Range query contests
- Dynamic programming with BIT

Skills to Practice:

- Coordinate compression
- 2D Fenwick Trees
- Range updates with difference arrays

Further Learning

Advanced Topics:

- Persistent Fenwick Trees
- Parallel/Concurrent Fenwick Trees
- Range updates with range queries
- Fenwick Tree on trees (Heavy-Light Decomposition)
- Offline query optimization

Resources:

- CP-Algorithms: Binary Indexed Tree tutorial
- Topcoder: Range query structures
- Codeforces: Educational rounds on BIT
- "Competitive Programmer's Handbook"

Related Structures:

- Segment Trees
- Sqrt Decomposition

Thank You!

Questions?

Fenwick Trees: Simplicity meets Efficiency