FERNWICK TREES

Introduction

Fenwick Trees, also known as Binary Indexed Trees (BITs), are a data structure that provides efficient methods for querying and updating prefix sums of a list of numbers. They are particularly useful in scenarios where you need to perform multiple updates and prefix sum queries on an array of numbers.

Key Concepts

1. Prefix Sum: The sum of elements from the start of an array up to a given index.

2. Efficient Queries and Updates: Fenwick Trees allow both prefix sum queries and updates in O(\log n) time, where n is the number of elements.

Structure

A Fenwick Tree is a binary indexed tree, where each node at index i stores information about a segment of the array. The key idea is to store cumulative frequency or prefix sums in such a way that both updates and queries can be performed efficiently.

Construction

To build a Fenwick Tree, you initialize an array `BIT` (Binary Indexed Tree) of the same size as the input array, with all elements initially set to zero. The tree is built by updating the BIT array with the values from the input array.

Operations

1. Update Operation: To update the value at a specific index and propagate this change through the tree.

2. Query Operation: To compute the prefix sum up to a specific index efficiently.

Algorithms

Update Operation

To update the value at a given index i:

1. Increment the value at `arr[i]` by a given delta.

2. Update the BIT array by adding the delta to the appropriate indices.

The update process involves moving to the next index that includes the updated value. This is done using the expression i + (i \& -i).

Algorithm:

```
def update(bit, n, index, delta):
    while index <= n:
        bit[index] += delta
        index += index & -index
```

Query Operation

To find the prefix sum up to a given index i:

1. Initialize a variable to store the sum.

2. Add the value at `BIT[i]` to the sum.

3. Move to the parent node using the expression i - (i \& -i).

Algorithm:

```
def query(bit, index):
    sum = 0
    while index > 0:
        sum += bit[index]
        index -= index & -index
    return sum
```

Example Usage

Let's say we have an array `[3, 2, -1, 6, 5, 4, -3, 3, 7, 2, 3]` and we want to construct a Fenwick Tree to efficiently perform updates and prefix sum queries.

```
def build_bit(arr):
    n = len(arr)
    bit = [0] * (n + 1)
```

```
    for i in range(n):
        update(bit, n, i + 1, arr[i])
    return bit

arr = [3, 2, -1, 6, 5, 4, -3, 3, 7, 2, 3]
bit = build_bit(arr)

print("Prefix sum up to index 5:", query(bit, 5))  # Output should be 15
update(bit, len(arr), 3, 2)  # Increment arr[2] by 2
print("Prefix sum up to index 5 after update:", query(bit, 5))  # Output should be 17
```

Applications

1. Range Sum Queries: Efficiently answering queries about the sum of elements in a specific range.
2. Frequency Tables: Managing dynamic frequency tables and cumulative frequency tables.
3. Inversion Count: Counting the number of inversions in an array, which is useful in sorting and order statistics.
4. 2D Range Queries: Extending the concept to 2D arrays for querying sub-matrix sums.

Advantages

- Efficiency: Both updates and prefix sum queries are performed in O(\log n) time.
- Simplicity: The data structure is simple to implement and understand.
- Space-Efficient: Requires O(n) space for n elements.

Limitations

- Static Size: The size of the Fenwick Tree is fixed at the initialization and does not support dynamic resizing.
- Complex Operations: While prefix sum queries and point updates are efficient, more complex operations like range updates are not directly supported and require additional techniques.

Conclusion

Fenwick Trees are a powerful tool for scenarios involving dynamic updates and prefix sum queries. They strike a good balance between simplicity and efficiency, making them suitable for various algorithmic challenges in competitive programming and practical applications. Understanding and implementing Fenwick Trees can significantly optimize solutions that involve cumulative frequency tables and range queries.