<https://dev.to/wengao/fenwick-tree-vs-segment-tree-1edk>

When it comes to solving range-based queries on an array, two of the most commonly used data structures are Fenwick Tree and Segment Tree. Both data structures offer efficient solutions to range-based queries and can be used interchangeably in many cases. However, there are some key differences between the two that can affect their suitability for different use cases.

In this article, we'll take a closer look at both data structures, their strengths and weaknesses, and when you might choose one over the other.

**What is a Fenwick Tree**

Fenwick Tree, also known as Binary Indexed Tree (BIT), is a data structure that allows efficient querying and updating of prefix sums in an array. The key feature of a Fenwick tree is that it uses a binary representation of indices to compute prefix sums. It has a space complexity of O(N) and a time complexity of O(log N) for both range queries and point updates.

BIT is easy to implement and is particularly useful for solving problems that require cumulative frequency or sum queries, such as finding the sum of all elements in a given range. It works by storing the cumulative frequency of elements up to a certain index i in the i-th position of the BIT array.

**How to implement a BIT**

To build a binary indexed tree, we start with an array of numbers A and create a new array T, where T[i] stores the sum of a range of elements from A that starts at i and includes the 2^(lowbit(i)) previous elements (where lowbit(i) is the least significant bit that is 1 in the binary representation of i).

For example, let A = [1, 2, 3, 4, 5, 6, 7, 8]. For convenience consideration, we assume the array is 1-indexed. Then we can build a binary indexed tree T as follows:

T[1] = A[1] = 1

T[2] = A[1] + A[2] = 3

T[3] = A[3] = 3

T[4] = A[1] + A[2] + A[3] + A[4] = 10

T[5] = A[5] = 5

T[6] = A[5] + A[6] = 11

T[7] = A[7] = 7

T[8] = A[1] + A[2] + A[3] + A[4] + A[5] + A[6] + A[7] + A[8] = 36

A typical implementation of BIT:

public class FenwickTree {

private final int[] tree;

public FenwickTree(int n) {

this.tree = new int[n];

}

public void add(int idx, int val) {

for (int i = idx; i < tree.length; i += lowbit(i)) {

tree[i] += val;

}

}

public int query(int idx) {

int sum = 0;

for (int i = idx; i > 0; i -= lowbit(i)) {

sum += tree[i];

}

return sum;

}

private int lowbit(int x) {

return x & -x;

}

}

If we want to initialize the BIT from an input array, we can do something like:

void someMethod(int[] nums) {

FenwickTree tree = new FenwickTree(nums.length + 1);

for (int i = 0; i < nums.length; ++i) {

tree.add(i + 1, nums[i]);

}

// use the tree to do further queries and updates.

// ...

}

**What is a Segment Tree**

Compared to Fenwick Tree, Segment Tree is a more general-purpose data structure used to efficiently perform range-based queries and range updates on an array. It is an extension of the concept of Fenwick Tree and can handle more complex queries/updates. Like a Fenwick tree, a segment tree is also based on the divide-and-conquer approach, where a problem is recursively divided into subproblems until a base case is reached. Segment Tree has a space complexity of O(N) and a time complexity of O(log N) for both range queries, point updates and range updates.

If you are interested in more details regarding the segment tree, please refer to the following three articles:

[Efficiently Implement a Basic Segment Tree](https://wengao.hashnode.dev/efficiently-implement-a-basic-segment-tree)

[Segment Tree with Lazy Propagation](https://wengao.hashnode.dev/segment-tree-with-lazy-propagation)

[Utilize Segment Tree to Solve Problem](https://wengao.hashnode.dev/utilize-segment-tree-to-solve-problem)

**Comparison**

Fenwick Tree is generally simpler to implement than Segment Tree and can be used to solve problems that only require cumulative frequency or sum queries. It is also more space-efficient, with a space complexity of O(N) compared to O(2N) for Segment Tree (**NOTE**: If we use a physical binary tree structure to implement, the space complexity is O(2N); If we use a bounded array to implement, the space complexity is O(4N). For more details, please refer to: [Efficiently Implement a Basic Segment Tree](https://wengao.hashnode.dev/efficiently-implement-a-basic-segment-tree)).

Segment Tree, on the other hand, is more general-purpose and can be used to solve a wider range of problems. It is also more flexible in terms of the operations it can perform, with support for a wider range of operations than Fenwick Tree. Especially, [Segment Tree with Lazy Propagation](https://wengao.hashnode.dev/segment-tree-with-lazy-propagation) can support range updates in O(log N) time, while Fenwick Tree only supports point updates.

Another important difference between the two is the time complexity of point updates. While both data structures have a time complexity of O(log N) for point updates, Fenwick Tree is generally faster in practice due to its simpler implementation and better cache performance.

Put simply, if a problem can be solved by a Fenwick Tree, then it can certainly be solved by a Segment Tree as well. But the reverse is not necessarily true. However, due to the simpler implementation and faster point updates in practice, Fenwick Tree is preferable over Segment Tree if a problem can be solved by them.

**Application**

Now let's see how to use Fenwick Tree or Segment Tree to solve a practical problem.

**Problem**

Given an integer array nums, handle multiple queries of the following types:

Update the value of an element in nums.

Calculate the sum of the elements of nums between indices left and right inclusive where left <= right.

Implement the NumArray class:

NumArray(int[] nums) initializes the object with the integer array nums.

void update(int index, int val) updates the value of nums[index] to be val.

int sumRange(int left, int right) returns the sum of the elements of nums between indices left and right inclusive (i.e. nums[left] + nums[left + 1] + ... + nums[right]).

**Solution using Fenwick Tree**

public class NumArray {

private final int[] nums;

private final int[] tree;

public NumArray(int[] nums) {

this.nums = nums;

this.tree = new int[nums.length + 1];

for (int i = 0; i < nums.length; ++i) {

add(i + 1, nums[i]);

}

}

public void update(int index, int val) {

int delta = val - nums[index];

nums[index] = val;

add(index + 1, delta);

}

public int sumRange(int left, int right) {

return query(right + 1) - query(left);

}

private void add(int idx, int val) {

for (int i = idx; i < tree.length; i += lowbit(i)) {

tree[i] += val;

}

}

private int query(int idx) {

int sum = 0;

for (int i = idx; i > 0; i -= lowbit(i)) {

sum += tree[i];

}

return sum;

}

private int lowbit(int x) {

return x & -x;

}

}

Time complexity:

NumArray(int[] nums): O(N \* log N)

void update(int index, int val): O(log N)

int sumRange(int left, int right): O(log N)

**Solution using Segment Tree**

public class NumArray {

private final int[] tree;

private final int len;

public NumArray(int[] nums) {

this.len = nums.length;

this.tree = new int[nums.length \* 4];

this.buildTree(0, 0, nums.length - 1, nums);

}

public void update(int index, int val) {

update(0, 0, len - 1, index, val);

}

public int sumRange(int left, int right) {

return query(0, 0, len - 1, left, right);

}

private int query(int node, int start, int end, int left, int right) {

if (right < start || end < left) {

return 0;

}

if (left <= start && end <= right) {

return tree[node];

}

int sum = 0;

int mid = start + (end - start) / 2;

if (left <= mid) {

sum += query(2 \* node + 1, start, mid, left, right);

}

if (right > mid) {

sum += query(2 \* node + 2, mid + 1, end, left, right);

}

return sum;

}

private void update(int node, int start, int end, int idx, int val) {

if (idx < start || end < idx) {

return;

}

if (idx == start && idx == end) {

tree[node] = val;

return;

}

int leftChild = 2 \* node + 1;

int rightChild = 2 \* node + 2;

int mid = start + (end - start) / 2;

if (idx <= mid) {

update(leftChild, start, mid, idx, val);

} else {

update(rightChild, mid + 1, end, idx, val);

}

tree[node] = tree[leftChild] + tree[rightChild];

}

private void buildTree(int node, int start, int end, int[] nums) {

if (start == end) {

tree[node] = nums[start];

return;

}

int leftChild = 2 \* node + 1;

int rightChild = 2 \* node + 2;

int mid = start + (end - start) / 2;

buildTree(leftChild, start, mid, nums);

buildTree(rightChild, mid + 1, end, nums);

tree[node] = tree[leftChild] + tree[rightChild];

}

}

Time complexity:

NumArray(int[] nums): O(N)

void update(int index, int val): O(log N)

int sumRange(int left, int right): O(log N)