Data Structure: Fenwick Tree

A Fenwick Tree, sometimes known as a Binary Indexed Tree, is an efficient data structure for updating elements and calculating prefix sums in an array. The array is represented as a binary tree, where the value of a node is equal to the sum of the numbers in the node's subtree. This allows operations to be performed by accessing only **O(logn)** nodes, where **n** is the size of the array.

Fenwick Trees are usually used to quickly query cumulative frequencies in an array, whilst being able to quickly update the array as well.

Implementation

Despite being trees, Fenwick Trees are usually implemented as an implicit data structure using an array. It is easier to consider the array as being 1-indexed, so in this implementation we will leave the 0th element of the array empty. Each element in the array contains the sum of all of the values in the array since its parent (remember this is still an implicit tree). A node's parent is found by removing the Least Significant Bit (LSB) from the node's index. For example, any node with an index **i** of a power of 2 has no parent, and so contains the sum of the first **i** elements in the array.

As our array is an implicit binary tree, its size should be a power of 2. However, we are not using the 0th element, and so its size should be 1 + a power of 2. This gives a Fenwick Tree a memory usage of **O(n)**.

The function to return the LSB of an integer is given below.

```
int SIZE = (1<<10) + 1;
int fenwick_tree[SIZE];

int LSB(int value) {
    return value & -value;
}
```

Querying

A Fenwick Tree can be queried to find the prefix sum of the first **n** elements. This can be done in **O(logn)** time.

Suppose we want to find the sum of the first 11 elements in our underlying array. We know that the 11th element of the Fenwick Tree array contains the sum of all of the elements in the underlying array since its parent. But what is its parent? 11 in binary is 1011_2, so its parent is 1010_2, or 10. The parent of the 10th element is thus 1000_2, or 8, which is a power of 2 and so doesn't have a parent. These nodes sum of the ranges, 11, 9-10 and 1-8 respectively. Therefore, to find the prefix sum of the first **n** elements, we must sum the 8th, 10th and 11th elements.

```
7  int get_prefix_sum(int[] arr, int index) {
8      int sum = 0;
9      while (index > 0) {
10         sum += arr[index];
11         index -= LSB(index);
12     }
13 }
```

Updating

What if we want to increment the 11th element of our underlying array instead? To update the 11th element, we need to modify elements 1011_2 (11), 1100_2 (12), 10000_2 (16), and every power of 2 up to the size of the array. This changes updates all of the ranges which include element 11, which are 11, 9-12, 1-16, 1-32, and so on.

Thus, updating an element in a Fenwick Tree can be done in **O(logn)**, where **n** is the size of the array.

```
14 void update(int[] arr, int index, int value) {
15     while (index < SIZE) {
16         arr[index] += value;
17         index += LSB(index);
```

```
    }
}
```

Adaptations

Fenwick Trees can be used to efficiently calculate more than just prefix sums in an array. Almost any function can be used in a Fenwick Tree. For example, they can be used to calculate the prefix product, or the prefix minimum. However, some of the following Fenwick Tree methods require a function that has an inverse. These methods do not work with functions such as minimum or maximum, but do work with functions such as addition or multiplication.

Instead of calculating a prefix sum, Fenwick Trees can also be used to calculate a range sum within the array. This can be achieved by calculating the prefix sum for the upper bound of the range, and subtracting the prefix sum for the lower bound of the range. The code below implements this in a more efficient manner. ***Note: This method requires the function to have an inverse.***

```
// returns the sum of elements from start + 1 to end
int get_range_sum(int[] arr, int start, int end) {
    int sum = 0;
    while (end > start) {
        sum += arr[end];
        end -= LSB(end);
    }
    while (start > end) {
        sum -= arr[start];
        start -= LSB(start);
    }
    return sum;
}
```

We can use this function to get or set the value of an element in the underlying array :

```
int get(int index) {
```

```
34          return range_sum(index, index + 1);
35  }
36
37  void set(int index, int value) {
38          add(index, value - get(index));
39  }
```