<u>Binary Index Trees</u>

A cumulative frequency array allows us to calculate the sum of the range of values in O(1), ***as long as there are no changes to the data once the queries start.***

But consider situations where we might change the value at an index in an array, then query for the sum of a range of values in the array, followed by some more changes and more queries. In this sort of situation, a cumulative frequency array would still give us O(1) query times, but it would take O(n) time to update after each change!!! (Basically, if we change one index in a cumulative frequency array, all other indexes above it would have to have this value added to it as well.) Here is a quick illustration:

Current cumulative frequency array:

Index	0	1	2	3	4	5	6	7	8
Value	2	4	4	4	6	8	11	13	17

Now, consider adding the value 2 to the data, recalling that index i stores the number of values less than or equal to i. The adjusted array is:

Index	0	1	2	3	4	5	6	7	8
Value	2	4	5	5	7	9	12	14	18

We had to edit each index 2 or greater, which would take O(n) for a cumulative frequency array of size n.

Thus, we want a new arrangement where both the query AND the update are relatively fast. The creative insight here is that perhaps if we store data in a different way, perhaps we can reduce the update time by quite a bit while incurring only a modest increase in query time. A binary index tree (also called a Fenwick tree), achieves just this. It achieves a O(lg n) time for both the update and the query.

<u>*What to Store in each Index*</u>
In a cumulative frequency array, array[i] stores $\sum_{k=0}^{i} x[k]$, where x[k] represents the original items. In a binary index tree (which is really just stored in an array), each index will store a sum of values from the original array as well, but the set of values summed at each index will be more complicated than in a cumulative frequency array.

<u>*Number of Values summed at index i*</u>
The value stored in index i of a binary index tree is based upon the binary representation of i, hence the name, binary index tree.

For this description, let x[i] represent the original values.

Each index will always store a sum of a set of values of x[i] that has a size that is a perfect power of 2. In particular, the number of values added to get the value stored in index i is 2^L, where L represents the place of the lowest 1 bit. For example if the binary representation of i is 10000100, then there are $2^2 = 4$ values summed at index i, since the least significant bit equal to one is in the 2^2 place, third from the right. If i in binary is 1010100000, then the corresponding value is the sum of 2^5 values in the original array.

__Which values are summed at index i__

Now that we know HOW many values of the original array are summed at index i, we must specify WHICH values comprised that sum.

The values that comprise the sum at index i are the *last* 2^L values of x[i]. Thus, we finally can write this mathematically:

BIT[i] = $\sum_{k=i-2^L+1}^{i} x[k]$, where L is the lowest one bit location as previously defined.

Here is a table of what is stored in the first 16 indexes of a binary index tree:

Index	Values Summed From Original Array
1	x[1]
2	x[1]+x[2]
3	x[3]
4	x[1]+x[2]+x[3]+x[4]
5	x[5]
6	x[5]+x[6]
7	x[7]
8	x[1]+x[2]+x[3]+x[4]+x[5]+x[6]+x[7]+x[8]
9	x[9]
10	x[9]+x[10]
11	x[11]
12	x[9]+x[10]+x[11]+x[12]
13	x[13]
14	x[13]+x[14]
15	x[15]
16	x[1]+x[2]+x[3]+…x[15]+x[16]

__Indexes to Change for an Update__

Consider an update to an arbitrary index x[i] of the original array. If you look at the table above, we can see that the total number of indexes that refer to x[i] is logarithmic in the size of the whole table. Consider x[5] - it's part of BIT[5], BIT[6], BIT[8] and BIT[16]. To get from 5 to 6, we add 1, the value of the lowest one bit in 5. To get from 6 to 8, we add 2, the value of the lowest one bit in 6. To get from 8 to 16, we add 8, the value of the lowest one bit of 8, and so forth.

Thus, the algorithm for adding D to the value of x[i] is as follows:

```
1. Let curIndex = i.
2. while curIndex < sizeof(BIT)
        2a. BIT[curIndex] += D
        2b. curIndex += valueOfLowestOneBit(curIndex)
```

Indexes to Add for a Query

Consider just supporting queries in the range 1 to i. (Any range query can be represented as the difference of two range queries of this time.) Let's consider summing the values x[1] + x[2] +… + x[13]. First we'd add BIT[13]. This just adds x[13]. Then we would add BIT[12], adding x[9] + x[10] + x[11] + x[12]. Finally we would add BIT[8], which would add x[1]+x[2]+x[3]+x[4]+x[5]+x[6]+x[7]+x[8].

In short, we add our current index we are at in the BIT array, and then subtract the value of the lowest one bit from the current index. Thus, the algorithm for the query sum of x[1] to x[i] is as follows:

```
1. Let curIndex = i.
2. Let sum = 0.
3. while curIndex > 0
        2a. sum += BIT[curIndex]
        2b. curIndex -= valueOfLowestOneBit(curIndex)
```

Java Implementation

The nice thing is that Java's Integer class has a method, lowestOneBit that returns the numeric value of the lowest one's bit of an integer. For example, Integer.lowestOneBit(24) returns 8, since the binary representation of 24 is 11000 and the value of the right most one is 8. So, the add method, which adds value to the given index of the original data looks like this:

```java
public void add(int index, long value) {
    while (index < cumfreq.length) {
        cumfreq[index] += value;
        index += Integer.lowestOneBit(index);
    }
}
```

Here is the sum method, which returns the sum of all items of the original array upto index:

```java
public long sum(int index) {
    long ans = 0;
    while (index > 0) {
        ans += cumfreq[index];
        index -= (Integer.lowestOneBit(index));
    }
    return ans;
}
```

Sample Problem: Candy

Consider that there are 100,000 candies, numbered 0 through 99,999. We start eating the candies in some random order. We might eat candy number 34,567, followed by candy number 12,980, etc. At any point however, we might be asked how many candies are still available from some range. For example, after eating several candies, we might be asked to determine the number of candies that aren't eaten in between candy number 12,000 and candy number 12,999, inclusive. If we had eaten candy number 12,980 already, then the answer to the query would be 999.

A binary index tree is the perfect data structure to allow us to update which candies have been eaten and answer many queries of this nature in sequence.

One way to solve the problem is as follows:

1) Start with an empty binary index tree.

2) Whenever an item is eaten add 1 to that specific slot in the binary index tree.

3) Whenever there is a query, you can query a range to see how many candies in that range were eaten. Just subtract this value from the total number of values in that range.

Sample Problem: ZigZag2

A zigzag sequence is one that goes up and down. For example, 3, 19, 17, 14 is a zigzag sequence as is 17, 13, 16, 1, 2, 1, 5, 4. Given an arbitrary sequence of length up to 10^5, determine the number of zigzag subsequences contained in the sequence of length 3 or greater. All of the values in the sequence are in between -10^9 and 10^9.

Consider just solving the problem for all zigzag sequences, regardless of their length. (Afterwards, we can figure out how to subtract out all the sequences we over-counted of lengths 1 and 2.)

First, note that we can compress the input data by noting how many unique items (call this m) there are and reassigning each one a number from 0 to m-1. For example, if the original sequence was 18, 3, -5, 14, 9, 3, 14, then the mapping would be f(-5) = 0, f(3) = 1, f(9) = 2 f(14) = 3 and f(18) = 4. Our mapped sequence would then be 4, 1, 0, 3, 2, 1, 3. The number of zigzag subsequences in this sequence is equal to the number of zigzag subsequences in the original, since we haven't changed any relative values.

Let this mapped input sequence be a[0], a[1], … a[n-1], with all values in the range from 0 to m-1. (Note that $m \le n$.) Let numUpSeq[i] store the number of zigzag subsequences that end at the value i that are currently going up and let numDownSeq[i] store the number of zigzag sequences that end at the value i, currently going down.

Consider we arrive at some number, x after processing some part of the input. We want to know how many sequences we can tack x onto. Well, there are two options:

1) Tacking x onto a sequence that is currently heading up that ends at x+1 or higher.
2) Tacking x onto a sequence that is currently heading down that ends at x-1 or lower.

The new sequences in the first set are built by adding numSeqUp[x+1] + numSeqUp[x+2]+…numSeqUp[m-1], a query perfect for a binary index tree. These new sequences are going down and should be added to numSeqDown[x].

The new sequences in the second set are built by adding numSeqDown[0]+numSeqDown[1]+…+numSeqDown[x-1]. These new sequences are going up and should be added numSeqUp[x].

At the end of running the Binary Index Trees for numSeqUp and numSeqDown, the sum of ALL the values in both trees represents all the unique zigzag subsequences we saw.

To solve the given problem though, we have to subtract all subsequences of lengths 1 or 2. Sequences of length 1 are easy: there is one of these for each number for both binary index trees. For sequences of length 2, we need keep a third binary index tree running just of each item, one at a time. When I get to an item x, I just want to know how many items prior to x were not equal to x (so the sum of the items above x and below x). All of these form valid zigzag subsequences of length 2. Here is the critical code to solve the problem, after the input value have been remapped (stored in nums):

```
long sub = 0;
bit up = new bit(index, MOD);
bit down = new bit(index, MOD);
bit seqBit = new bit(index, MOD);

for (int i=0; i<n; i++) {

    long prevUp = up.above(nums[i]);
    long prevDown = down.below(nums[i]);
    down.add(nums[i], prevUp+1);
    up.add(nums[i], prevDown+1);

    sub = (sub+2+ seqBit.above(nums[i]) + seqBit.below(nums[i]))%MOD;
    seqBit.add(nums[i], 1);
}

long res = (up.all() + down.all() - sub + MOD)%MOD;
```

Note: the methods above and below return the sum of items above and below the input value to the methods, respectively.