Binary Index Trees

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.

What to Store in each Index

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.

Number of Values summed at index i

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:

```
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:

```
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: Counting Inversions

An inversion in an array all pairs of indexes i and j such that i < j and array[i] > array[j]. (Essentially, the pair is out of order.) Consider the problem of counting the total number of inversions in an array. Here is a sample array:

3, 9, 2, 8, 7, 1, 4, 5, 6

For each value j > 0, we want to know how many values in the array before it are greater than array[j].

For 9, there are no values before it greater than it.
For 2, there are 2 values before it (3 and 9) greater than it.
For 8, there is 1 value before it (9) greater than it.
For 7, there are 2 values before it greater than it.
For 1, there are 5 values before it greater than it.
For 4, there are 3 values before it greater than it.
For 5, there are 3 values before it greater than it.
For 6, there are 3 values before it greater than it.
This is a total of 2 + 1 + 2 + 5 + 3 + 3 + 3 = 19 inversions.

Normally, using a straight-forward algorithm, this would take $O(n^2)$ time for an array with n elements. We would like to speed up the inner for loop. Instead of running through each item array[0] through array[j-1] and seeing how many are greater than array[j], we'd like to answer this question without looking at each value.

A binary index tree helps us as follows:

1. Start with an empty binary index tree.
2. As we loop through each element in the array, query the BIT to see the sum of the BIT from array[i]+1 to MAX. This is the number of elements greater than array[i] that come before it.
3. Now, after adding that to our total number of inversions, add 1 to array[i] in the BIT, indicating that for the future, there is a number array[i] that has already been processed.

Here is a trace through of the new algorithm with the same array, included for convenience:
3, 9, 2, 8, 7, 1, 4, 5, 6

So, we start with 0 inversions.
When we process 3, there are no sum greater than 3 in our BIT.
Then we add 1 to index 3 of our BIT.
When we process 9, there is no sum greater than 9 in our BIT.
Then we add 1 to index 9 of our BIT.
When we process 2, there is a sum of 2 for values greater than 2 in our BIT.
Then we add 1 to index 2 of our BIT.
When we process 8, there is a sum of 1 for values greater than 8 in our BIT.
Then we add 1 to index 8 of our BIT.
When we process 7, there is a sum of 2 for values greater than 7 in our BIT.
Then we add 1 to index 7 of our BIT.
When we process 1, there is a sum of 5 for values greater than 1 in our BIT.
Then we add 1 to index 1 of our BIT.
When we process 4, there is a sum of 3 for values greater than 4 in our BIT.
Then we add 1 to index 4 of our BIT.
When we process 5, there is a sum of 3 for values greater than 5 in our BIT.
Then we add 1 to index 5 of our BIT.
When we process 6, there is a sum of 3 for values greater than 6 in our BIT.
Then we add 1 to index 6 of our BIT.

Sample Problem: Bread Sorting

In this problem, you are attempting to reorder some pieces of bread from one ordering (one permutation) to another ordering (another permutation). Unfortunately, you can't just swap any pieces of bread. The only operation you are allowed is on a group of 3 consecutive pieces of bread. If three pieces of bread are originally in the order of (a, b, c), then you can use your paddle to cyclically rotate them to the setting (c, a, b). The problem is to determine, if it's possible or not to transform the row of bread from the given starting ordering to the ending ordering.

The key observation in this problem is that when you do the exchange shown above, exactly two pairs of elements "swap" order. So a → c becomes c → a, and b →c becomes c → b. Regardless of whether a < c or a > c, this means that each paddle operation will change the number of inversions by +1 or -1 with regards to a and c, and also +1 or -1 with regards to b and c. Thus, the total number of changes in inversions must be either -2, 0 or 2. In short, when limited to using the paddle for sorting, the ***parity*** of the number of inversions stays the same. Thus, the solution is as follows: count the number of inversions in both arrays via a BIT as previously discussed. Then, the transformation is possible if and only if both counts have the same parity.

Note: Here is an informal proof that we can always make the transformation if the parities of the number of inversions of both arrays is the same:

For example, for the first sample input:

1 3 4 2
4 1 3 2
4 2 1 3
4 3 2 1

Basically, if the inversion parity holds, we should be able to get each item into place from left to right, except for the last three, because we have enough freedom to shuffle things around. Then, when we get to the last three, since the inversion count holds, one of the three cyclic rotations will be the correct match.

Sample Problem: Corgi Houses

This problem is the canonical straight-forward BIT problem. In the problem, there are upto 1,000,000 houses in a row. On each day, either ***c*** corgis are brought to house ***k***, or, we must answer a query as to how many corgis are in houses ***i*** through ***j***, inclusive. Notice that these are precisely the operations a Binary Index Tree performs in O(lg n) time. Thus, for each day that a corgi is brought to one house, we add ***c*** to index ***k.*** Whenever we must answer a query, the query is precisely the form of a query a Binary Index Tree is fit to answer.

Sample Problem: Longest Increasing Subsequence

In short, we want to count the number of increasing subsequences of length k ($k \leq 10$) on an array of size upto 50,000, where all the integers are positive and no greater than 100,000.

Consider being at index i, processing the value v, looking to see how many increasing subsequences of length k end with v at index i. The subproblem we would want to know the answer to is:

How many subsequences of length k-1 end at index i-1 or less, with value v-1 or less?

Imagine a Binary Index Tree, where we store the number of subsequences of length k-1 and at each value from 1 to 100,000 we store HOW MANY SUBSEQUENCES of that length end at that value. **THEN, we could get the answer to our question with a single Binary Index Tree query.**

The key observation here though is that in order for that Binary Index Tree to work, it would have had to have gotten its answers from a Binary Index Tree which stored information about subsequences of length k-2, and so forth.

So, the trick to this problem is that we have to store k Binary Index Trees, one for each possible length of subsequence. We build answers for the Binary Index Tree at index z from the Binary Index Tree at index z-1, so first we get information about the number of subsequences of length 1, then length 2, etc.

Since we want the Binary Index Tree to represent the state of the array only to the left, we are forced to do our outer loop in index i, the index into the original array. The inner loop has to go through the length of the sequence, but to make sure we don't build off the same value more than once, this loop should run backwards.