Fenwick Trees

Hello, and welcome to competitive programming. Today we are going to go over a data structure called a Fenwick Tree, which is a good data structure for storing cumulative ranges.

Objectives

Your objectives are to be able to implement a Fenwick tree.

Motivating Example

Suppose have a sequence of numbers and we want to keep track of their frequency. For instance, maybe we have a set of ratings from 1 to 5, or some exam scores from 1 to 10. I'll borrow the Competitive Programming book's example to start. Here are 11 exams scores ranging from 1 to 10. We don't care about individual exams here, what we want is to ask questions like "how many students scores 2 points".

(next)

We can build an array quickly where the index into the array is the particular score we are interested in and the value is the count. So, index 6 has value 3, because three students got six points on the exam.

Now let's suppose we want to store cumulative frequencies. This allows us to ask questions like "how many students got 5 or fewer points", or even "how many students got between 4 and 6 points" if we use two lookups.

(next)

That too, is pretty simple, and takes only a linear scan to make the cumulative array.

So here, index six has value 7. If we want to know how many people got from 4 and 6 points, we take index 6 and subtract index 3, to get 6. There were three 6's, two 5's, and one 4.

So far, so good. But what if we have to update this? Suppose we discover another exam, this one getting a score of three.

(next)

In this case, we have to update the three index, which is fast enough, but the cumulative array will need an order n update.

We need a faster way to make these kinds of updates.

Finwick Tree

The problem with the cumulative array is that each element carries in it the sum of its own index and all the indices previous to it. Each element has a linear amount of responsibility.

In a Fenwick tree, we change things by having nodes be responsible for only part of the range.

We want our nodes to have a logarithmic amount of responsibility. Here's how we will do that. Half of the nodes will store only their own index's count in them. They will be only responsible for themselves. Half of the nodes that are left will store the sum of their own index plus the index right before it. Then, half of the nodes that are left after that will store four nodes worth of summation, and so on and so forth.

(next)

The way we will pick which nodes is by looking at the lower order bit in the binary representation of the index.

Here's the array again. I've put the indices on the bottom this time, and also the binary representation.

Notice how the half of the nodes — every other node — has a low order bit with a 1. So let's start populating the Fenwick tree: every other node now has the data in it.

(next)

Now lets look at the elements with low order bits 1 0 . That would be index 2, 6, and 10 here. We will make them carry the sum of themselves and the node to the right.

To keep the node levels visually separate, I'm going to draw the array in multiple levels, but really they are one array. Lets fill in the Fenwick Tree elements with those counts. It will look like this.

(next)

I've marked the elements in bold that contribute to the sums.

Now we are going to handle elements with low order bits 1-0-0. The only element like that is 4 here. Try to visualize what will happen with this structure.

(next)

Here it is. At this point you can see how this array will behave like a tree. One node left: what will it's value be?

(next)

The final node is 8, and has value 10.

Queries

To make a query, we have to use the bit pattern of the index n. Here's how to do it. First, you look up n in the Fenwick tree as if it were a normal array.

Next, we discard n's least significant 1 and repeat the process until n becomes zero.

Here's an example. Suppose we want to the sum of the values up to and including 5.

(next)

First we look up five itself. The value in the Fenwick Tree of five is two. Now we drop the least significant one from n to get a new n of four.

(next)

The value in the tree for index four is also two, giving us a total value of four. If you look at the values in the original array, you see we have 0-1-0-1-2, which sums to four.

Let's do another example: this time let's look up 10.

(next)

The Fenwick Tree index 10 has value 1. Dropping the least significant one gives you a new n of 8.

(next)

The Fenwick Tree index 8 has value 10. This gives us a total value of 11.

You might want to try a few examples of this yourself to be sure you understand the process.

Updating follows a slightly different pattern. We need to update all the nodes that use an updated index as part of its sum.

We can do this by *adding* the least significant one to the index each time and performing the update. This makes sure we get all the nodes that cover a particular index.

So for example, if I wanted up update 5, for instance, I would need to access 5, then 6, then 8.

So.. how to get the least significant one?

(next)

That turns out to be very easy. This function: LSOne, for least significant one, works by doing a binary and with the number and it's negation.

That's it for Fenwick trees. See you in class.

Next
Transcript — Games →