Binary Indexed Trees

Ryan Jian, Sreenath Are

Octobert 4, 2013

1 Introduction

A binary indexed tree (BIT), also known as a Fenwick tree, is a data structure used to efficiently calculate and update cumulative frequency tables, or prefix sums. BITs typically only show up in Gold problems, however they could start appearing more often in Silver.

The problem can be defined as follows: Given an array a of size N and some associative binary operation + that is not necessarily addition (although we will assume it is throughout the lecture for the sake of simplicity), we want to support the following two operations:

1. *Query*(i): running total of all the frequencies from 0 to i in f $(f[0] + [1] + f[2] + ... + f[i])$.
2. *Update*(i, x): add x to $f[i]$.

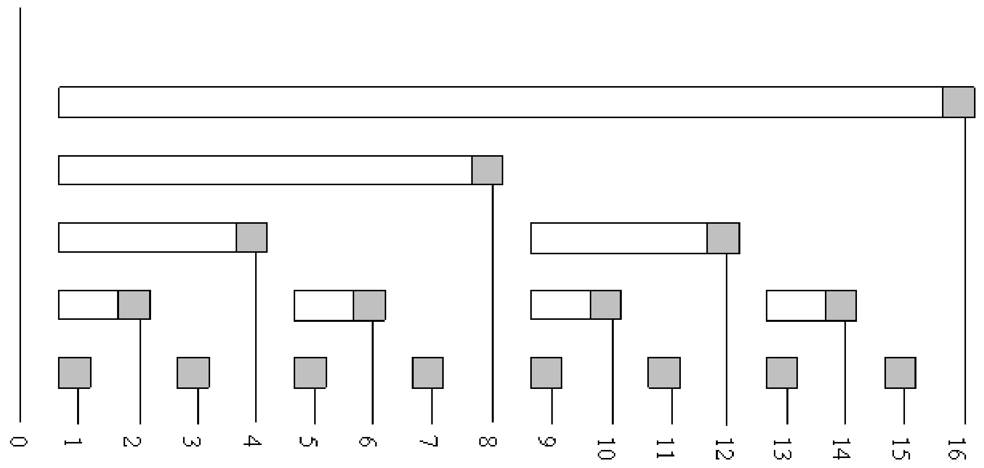
The naive solutions to this problem have time complexity $O(N)$ for query and $O(1)$ for update or vice versa. Better solutions use data structures, such as segment trees or BITs, that more evenly balance the time complexity between the two different operations and support both in $O(\log N)$.

2 General Overview

The central idea behind a BIT is that a cumulative frequency can be represented as the sum of a number of non-overlapping sub-frequencies, similar to how an integer can be represented as the sum of powers of two.

As a result, we'll define $tree[i]$ to be the sum of sub-frequencies from index $i - 2^r$ to i, where r is the index of the rightmost binary digit of i that is a 1.

Binary: $1011 \rightarrow 1010 \rightarrow 1000 \rightarrow 0000$
Decimal: $11 \rightarrow 10 \rightarrow 8 \rightarrow 0$



In order to actually find and subtract the rightmost digit that is a 1 in an efficient manner, we subtract the logical and of a number and its two's complement. To see why this works, consider an index i of the form $a1b$, where a is a binary sequence, and b is a binary sequence consisting of only zeros. Any nonzero index i has to be of this form.

$$-i = \overline{a1b} + 1 = \bar{a}0\bar{b} + 1 = \bar{a}0\overline{(0...0)} + 1 = \bar{a}0(1...1) + 1 = \bar{a}1b$$

$$\begin{array}{r} a1b \\ \&\bar{a}1b \\ \hline (0...0)1(0...0) \end{array}$$

3 Query

To implement $Query(i)$, we need to add up all the sub-frequencies that make up the ith cumulative frequency. Thus, we can just use the aforementioned indexing method and add up the subsets as we go down the tree.

```
def Query(i):
    index = i
    sum = 0
    while index > 0:
        sum += tree[index]
        index -= (index & -index)
    return sum
```

We can see that the time complexity of this operation is $O(\log N)$, because it runs based on the number of 1 bits in a given index, which is at most $\log N$.

Currently we can only query the sum of an interval $[0..i]$. To calculate the sum for an interval $[i..j]$ we can subtract $[0..i-1]$ from $[0..j]$.

4 Update

To implement $Update(i, x)$, we need to increase the value at the ith subfrequency and propogate that change back up the tree. It turns out that this can be done by repeatedly adding the rightmost 1 bit to i until we've exceeded the bounds of the tree.

```
def Update(i, x):
    index = i
    while index <= N:
        tree[index] += x
    index += (index & -index)
```

Update's time complexity is also $O(\log N)$ using the same argument in Query.

5 Scaling

Sometimes it is desirable to scale the tree by a constant factor. For addition, this is done by scaling each of the individual sub-frequencies in the tree and valid because of the distribute property. The time complexity is $O(N)$ and the implementation is trivial.

If our tree consists of non-negative frequencies, it is possible to to find an index with a given cumulative frequency using binary search.

6 Extension to 2D

BITs can also be extended to 2 dimensions and above. For 2 dimensions, query returns number of points in a rectangle $(0, 0)$, (x, y), and update will increase the number of points at (x, y). The actual implementation will consist of a BIT of BITs and the code is very similar to 1D.

7 Comparison to segment trees

We'll be perfectly honest about this. Anything that can be done with a BIT can be done with a segment tree since both store cumulative quantities for a number of sub-intervals.

However, there are things a segment tree can do that a BIT cannot. Calculating the cumulative sum for the interval $[i..j]$ involves taking the difference between $[0..j] - [0..i - 1]$, implying that the associative binary operation must be invertible. Non-invertible functions like $max()$ and $min()$ cannot be used. A segment tree on the other hand computes the cumulative quantity by combining smaller disjoint intervals so the same restriction does not hold.

Knowing these shortcomings, why would we ever want to use a BIT? The main advantage is that BITs are easier to code, especially when implementing range updates. BITs also require half as much memory, and have a time complexity with a lower constant factor, although this is not as useful for our purposes.

8 Problems

1. (Brian Dean, 2012) FJ has set up a cow race with N $(1 \le N \le 100,000)$ cows running L laps around a circular track of length C $(1 \le L, C \le 25,000)$. The cows all start at the same point on the track and run at different speeds, with the race ending when the fastest cow has run the total distance of $L * C$. FJ notices several occurrences of one cow overtaking another. Count the total number of crossing events during the entire race.

2. (Brian Dean, 2011) Farmer John has lined up his N $(1 \le N \le 100,000)$ cows each with height H_i $(1 \le H_i \le 1,000,000,000)$ to take a picture of a contiguous subsequence of the cows, such that the median height is at least a certain threshold X $(1 \le X \le 1,000,000,000)$. Count the number of possible subsequences.

3. (SPOJ BRCKTS) Given a bracket expression of length N $(1 \le N \le 30,000)$ process M operations. There are two types of operations, a replacement, which changes the ith bracket into its opposite, and a check, which determines whether a bracket expression is correct.

4. (Codeforces ABBYY Cup 3.0 B2) The Smart Beaver has many beavers, numbered from 1 to N. He wants to shave the beavers using a special machine, the Beavershave 5000. The Beavershave 5000 will shave any subsequence of beavers whose numbers are strictly increasing from left to right in a single pass. Write a program to support the following operations: 1. Determine the minimum number of Beavershave 5000 passes to shave all the beavers with ids between x and y. 2. Swap beavers x and y.