Binary Indexed Trees

Nick Haliday, Ryan Jian

September 28, 2012

1 Introduction

A binary indexed tree, also known as a Fenwick tree, is a data structure used to efficiently calculate and update cumulative frequency tables, or prefix sums. Binary indexed trees 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, we want to at the very least support the following two operations:

1. Query(i) - sums all the frequencies from 1 to i in in f (f[1] + f[2] + ...f[i])
2. Update(i, x) - adds x to f[i]

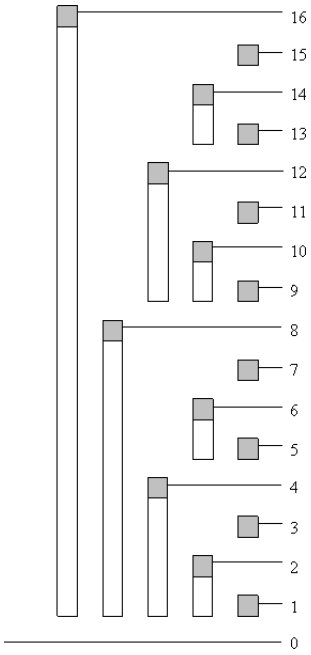
The naive solutions to this problem have time complexity $O(N)$ for query and $O(1)$ for update or vice versa. A binary indexed tree, on the other hand, more evenly balances the time complexity between the two different operations and supports both in $O(\log N)$. This problem can also be solved using other methods, for example Range Minimum Query, however a binary indexed tree is both easier to code, and requires less space.

2 General Overview

The central idea behind a binary indexed tree is that a cumulative frequency can be represented as the sum of a number of non-overlapping subfrequencies, 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 subfrequencies from index $i - 2^r + 1$ to i, where r is the index of the rightmost binary digit of i that is a 1. Notice that this means tree[i] is 1-indexed.

$$1011 \rightarrow 1010 \rightarrow 1000 \rightarrow 0000$$
$$11 \rightarrow 10 \rightarrow 8 \rightarrow 0$$



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.

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

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

3 Query

To implement Query(i), we need to add up all the subfrequencies 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 $\log N$.

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. 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 Other operations

Sometimes it is desirable to scale the tree by a constant factor. This is done by scaling each of the individual subfrequencies in the tree. This works because each subfrequency is determined by a series of linear operations meaning we can "distribute" the constant factor into each of the different subfrequencies. The time complexity is $O(N)$

```
def Scale(c):
    for each index i in the tree:
        tree[i] *= c
```

Keep in mind however, most problems using a binary indexed tree do not use these operations.

6 Extension to 2D

Binary indexed trees can also be used for 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 binary indexed tree of binary indexed trees (tree[i][j]) and the code is very similar to 1D.

7 Problems

1. (Brian Dean, 2012) Farmer John has set up a cow race with N ($1 \leq$ N $\leq$ 100,000) cows running L laps around a circular track of length C ($1 \leq$

L,C $\leq$ 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 LC. 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 ($1 \leq$ N $\leq$ 100,000) cows each with height H_i ($1 \leq$ H_i $\leq$ 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 ($1 \leq$ X $\leq$ 1,000,000,000). Count the number of possible subsequences.

3. (SPOJ MATSUM) A N X N matrix ($1 \leq$ N $\leq$ 1,024) is filled with numbers. FJ is analyzing the matrix, and he wants to answer Q queries of the following form:

   (a) SET x y num - Set the value of cell (x, y) to num ($0 \leq$ x,y $\leq$ N).

   (b) SUM x1 y1 x2 y2 - Find and print the sum of the values in the rectangle from (x1, y1) to (x2, y2), inclusive. You may assume that x1 $\leq$ x2 and y1 $\leq$ y2.

4. (SPOJ KPMATRIX) Given a matrix of integers with N rows and M columns ($1 \leq$ N,M $\leq$ 250), find the number of submatrices of the given matrix whose sum is between A and B ($-10^9 \leq A \leq B \leq 10^9$)

5. Arithmetic coding