**Binary Indexed Trees**

By Peter M. Fenwick

1. First used for data compression.
2. In contests, often used for storing frequencies and Manipulating cumulative frequency tables.

**Problem**: There are n boxes that undergo following queries:

1. Add marble to box.
2. Sum marbles from box k to box l.

* Naïve solution:

O(1) – query 1, O(n) – query 2.

For m queries: query 2 – O(m \* n).

* Using some data structure (RMQ) we can solve this problem with worst case time complexity of O(m\*log(n)) time.
* Another approach is Binary Indexed tree O(m \* log(n)).

**BIT :**

* Easier to code.
* Less memory space than RMQ.

**BIT: Binary Indexed Tree**

MaxIdx – Max. Index which will have non-zero frequency.

f[i] – frequency.

c[i] – cumulative frequency.

Tree[i] – sum of frequency stored at index i.

num¯ - complement of integer num. 0->1, 1->0

f[0] = 0, c[0] = 0, tree[0] = 0.

**Basic Idea:**

Each integer can be represented as a sum of power of 2. In the same way, a cumulative frequency can be represented as a sum of sets of sub frequencies.

In our case, each set contains some successive no. of non-overlapping frequencies.

r – position of least significant non-zero bit of idx.

Tree[idx] holds the sum of frequencies for indices (idx – 2 ^ r + 1) through idx.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **f** | 1 | 0 | 2 | 1 | 1 | 3 | 0 | 4 | 2 | 5 | 2 | 2 | 3 | 1 | 0 | 2 |
| **c** | 1 | 1 | 3 | 4 | 5 | 8 | 8 | 12 | 14 | 19 | 21 | 23 | 26 | 27 | 27 | 29 |
| **Tree** | 1 | 1 | 2 | 4 | 1 | 4 | 0 | 12 | 2 | 7 | 2 | 11 | 3 | 4 | 0 | 29 |

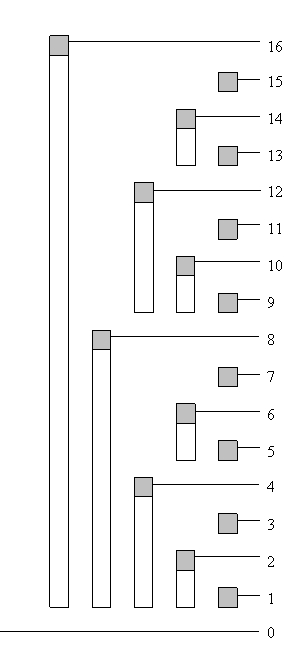


Image 1.3 – tree of responsibility for indices (bar shows range offrequencies accumulated in top element)

# Suppose you want to find the cumulative frequency at index 13, i.e. the sum of first 13 frequency .

c[1101] = tree [1101] + tree[1101] + tree[1000].

**# Isolating the last Bit :**

Note: For the sake of brevity, we will use “the last bit” to refer to the least significant non-zero bit of the corresponding integer.

* The algo for BIT require extracting the last bit of a number, so we need an efficient way of doing that.

Let num be an integer. We will now show how to isolate the last bit of num.

In binary, num = a1b where b = (00…0).

-num = (a1b)-1 + 1 = a-0b- + 1 = a–1b , b = all zeros , b - = all 1’s.

Now, (num & -num) = last non-zero bit.

int read(int idx) {

int sum = 0;

while(idx < 0) {

sum += tree[idx];

idx -= (idx & -idx);

}

return sum;

}

O(log(MaxIdx))

**# Updating tree**

**# Read actual frequency at a position**

* One approach is to maintain an additional array. In this array, we separately store the frequency for each index.

Reading or updating takes O(1) time.

Memory space is linear.

* First, the frequency at index idx can be calculated by calling the function read twice

f[idx] = read(idx) – read(idx-1).

1. \* O(log(n)).

* There is a different approach that has lower running time complexity than invoking read twice, lower by a constant factor.

**Approach:**

The main idea behind this approach is motivated by following observation. Assume that we want to compute the sum of frequency b/w two indices. For each 2 indices, consider the path from index to root, these two paths meet at some index (at latest at 0), after which point they overlap. Then, we calculate the sum of frequencies along each of those two paths until they meet and subtract those two sums. In that way we obtain the sum of frequencies b/w that 2 indices.

Algo:

Let x be an index and y = x – 1, we can represent y as a0b, b = (11…1)

x = a1b-, b- = (00…0)

Now we consider the first iteration of read applied to x.

Algo remove the last bit of x, hence replacing x by z = a0b-

y 🡪 z a0b 🡪 a0b- , at that point we stop.

int read\_single(int idx)

{

int sum = tree[idx];

int z = idx – (idx & -idx);

idx --;

while(idx != z) {

sum -= tree[idx];

idx -= (idx & -idx);

}

return sum;

}

For eg. Frequency at idx = 12.

Z = 12 – (12 & -12) = 8

Sum = 11

Y = 11(1011) 🡪 10(1010) 🡪 8(1000)

11 – 2 = 9 9 - 7 = 2

**# Scaling the entire tree by a constant factor.**

**# Finding index with a given cumulative frequency.**

Consider a task of finding an index which corresponds to a given cumulative frequency i.e. the task of performing an inverse operation of read.

A naïve and simple way to solve the task is to iterate through all the indices, calculate their cumulative frequency and output the index. (In case of negative frequency).

* In case of negative frequencies, it is only Known solution.
* Non-negative, we can use an algo. that sums in logarithmic time, modification of binary search.

If the tree exists more than 1 index with same cumulative frequency, this procedure will return some of them

Bitmask - Initially, it is greatest bit of MaxIdx.

Bitmask stores the current interval that should be searched.

int find(int cumfre) {

int idx = 0;

while(bitmask != 0) {

int tIdx = idx + bitmask;

bitmask >>= 1;

if(tIdx > MaxIdx)

continue;

if(cumfre > tree[tIdx]) {

idx = tIdx;

cumfre -= tree[tIdx];

}

}

if(cumfre != 0) return -1;

else return idx;

}

**# 2D BIT**

BIT can be used as multidimensional Data structure. Suppose you can have a plane with dots(with non-negative coordinates).

There are 3 queries at your disposal:

1. Set a dot at (x,y).
2. Remove the dot from (x,y).
3. Count the number of dots in rectangle (0,0) to (x,y) – where (0,0) is down left corner, (x,y) is up-right corner and sides are parallel to x-axis & y-axis.

O(m \* log(max\_x) \* log(max\_y))

Tree [max\_x][max\_y]

Updating indices of x-coordinate is same as before.

If we are setting/removing dots(a, b). we will call **update(a, b, 1) / update(a, b, -1).**

void update(int x, int y, int val) {

while(x <= max\_x) {

update\_y(x, y, val);

x += (x & -x);

}

}

void update\_y(int x, int y, int val) {

while(y <= max\_y) {

tree[x][y] += val;

y += (y & -y);

}

}

**# Lazy Modification**

So far we have presented BIT as a structure which is entirely allocated in memory during the initialization. An advantage of this approach is that accessing tree[Idx] requires a constant time. On the other hand, we might need to access only tree[idx] for a couple of different values of idx.

E.g. log n different values, while we allocate much larger memory. This is especially apparent in the cases when we work with multidimensional BIT.

To alleviate this issue, we can allocate the cells of a BIT in a lazy manner i.e. allocate when they are needed.

For instance, in the case of 2D, instead of defining BIT tree as a 2D array, in C++ we could define it as map< pair <int, int>, int>. Then accessing the cell at position (x, y) is done by invoking tree[make\_pair(x, y)].

Since every query visits O(log(max\_x) \* log(max\_y)) cells, if we invoke q queries the no. of allocated cells will be O(q \* log(max\_x) \* log(max\_y)).

* Accessing (x,y) requires logarithmic time in the size corresponding map structure, compare to only constant time previously.

So, by taking a logarithmic factor in the running time we can obtain memory-wise effecient data structure that per query uses only O(log(max\_x) \* log(max\_y)) memory in 2D case.

Or only O(log( maxIdx)) memory in 1D case.