Binary Indexed Trees

Daniel Wisdom

November 30 2018

1 Introduction

A Binary Index Tree (BIT), also known as a Fenwick Tree, is used for range sums (usually). Namely, a BIT can do element updates and prefix sums ($a[1] + a[2] + ... + a[i]$; we one-index BITs for implementation-specific reasons) in $O(\log n)$. This is a tradeoff between a $O(n)$ update/$O(1)$ query prefix-sum solution and the $O(1)$ update/$O(n)$ query naive solution.

BITs are very useful, especially for their simple implementation.

2 BITs

1	4	6	-2	3	-10	2	2	0	12	4	1	-1	6	5	2
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 1: A sample array.

BITs rely on the idea that an integer can be decomposed into powers of two. Given an index i, we can find these powers of two by writing i in binary. Then, we keep turning off the lowest bit until we reach zero. Say we want to find the prefix sum $a[1] + a[2] + ... + a[14]$:

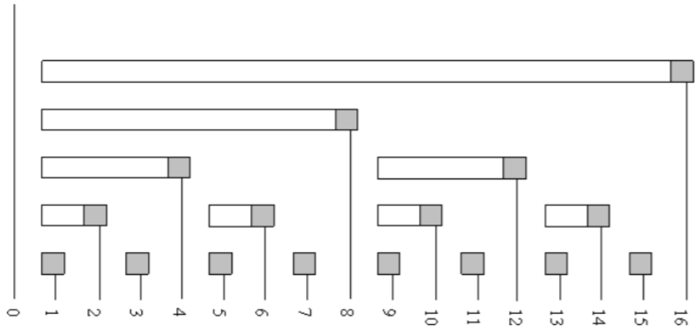
$$14 \rightarrow 1110 \rightarrow 1100 \rightarrow 1000 \rightarrow 0$$

How do we find the prefix sum with this?

Say we just went from 1110 → 1100. We just jumped from index 14 → 12. We can add the elements with indices 13 and 14 to a running sum, then recur on 12:

$$\texttt{1110}\ (14) \xrightarrow[\texttt{add}\ a[13]+a[14]]{} \texttt{1100}\ (12) \xrightarrow[\texttt{add}\ a[9]+...+a[12]]{} \texttt{1000}\ (8) \xrightarrow[\texttt{add}\ a[1]+...+a[8]]{} 0$$

Notice that every "step" (1110, 1100, and 1000), there's a unique range of indices denoted. That is, `1110` uniquely denotes indices `1101` and `1110`, or all numbers between the `1110` and $1 +$ (`1110` with the bottom bit removed). So we can map every number to a range of indices, and store the sum beforehand; `1110` stores $a[13] + a[14]$. See the illustration below.



2.1 Query

We discussed query above. But how do we find the lowest bit?

Taking advantage of the two's complement system ($-1 = 1...1111_2, -2 = 1...1110_2$ and so on), we can do this very easily. Say we're using $14 = 1110_2$. $-14 = 0010_2$ (with a bunch of ones in front). If we bitwise AND these two together, we get only the lowest bit set. This holds true in a general sense: let $i = (a1b)_2$, where a and b are parts of the binary number, and the one represents the lowest bit set. Then the negative is as follows: $-i = \tilde{}(a1b)_2 + 1 = \tilde{}a0\,\tilde{}b + 1$. But b must consist of only zeros, since it's after the lowest set bit. Therefore $\tilde{}b + 1 = 100...$ Thus, we get $-i = (\,\tilde{}a1b)_2$. Bitwise AND-ing with i, we clearly see that only the lowest bit is set.

A C++ implementation is shown below.

```
int query(int i) {
    int ans = 0;
    for(; i>0; i-=(i & -i))
        ans += a[i];
    return ans;
}
```

Clearly, to do range queries, we can subtract in the same way we do with regular prefix sums:

```
int range(int i, int j) {
    return query(j) - (i>1?query(i-1):0);
}
```

2.2 Update

To update (add a value v) at a given index i, we want to add the value to all segments "above" i. Here I mean "above" in the sense of the diagram above – all segments that contain i.

Let's take 9. The sequence for segments "above" 9 is:

$$\texttt{9}\ (1001) \rightarrow \texttt{10}\ (1010) \rightarrow \texttt{12}\ (1100) \rightarrow 16(10000).$$

Notice that we're simply adding the lowest bit every time (why?). Then for each index we visit, we add v to the value at this segment. Thus, the implementation is quite similar to query.

```
int update(int i, int v) {
    for(; i<=N; i += (i & -i))
        a[i] += v;
}
```

Note that for both update and query, we're only going through each bit once. Thus, the complexity is $O(\log n)$.

2.2.1 Range Updates

Range updates, where we add some number to all elements on $[l, r]$ are a bit more involved, but can also be done in $O(\log n)$. The idea is to keep two BITs. Remember, we one-index BITs.

Let's say we want to find a given prefix sum to index i (to find the range sum we can still subtract the prefix sums). To do this, we find all ranges that begin before i. Then, the answer is:

$$\sum_{ranges} max(i, r) * v - (l - 1) * v$$

where r is the right endpoint of a given range, l is the left, and v is the value. To calculate this, we can use two BITs. BIT1.query(w) will give the value of $a[w]$.

We will use BIT1.query(w)*w as a starting point for the prefix sum. There are two errors to account for:

- The range does not start at index 1. BIT1.query(w)*w assumes the active ranges start from 1. Update 3 fixes this.
- The range started and ended before w. BIT1.query(w)*w does not include any contribution from that range. Update 4 fixes this.

Specifically, here's how we'd update:

1. BIT1.update(l, v). All queries of BIT1 after (and including) l need to increase by v.
2. BIT1.update(r+1, -v). Queries of BIT1 past r should not be affected by this new interval. This cancels out Update 1 for everything past r.
3. BIT2.update(l, -(l-1)*v). BIT1.query(w)*w assumed the range started at 1. We subtract out (l-1)*v, the exact amount BIT1.query(w)*w over counted.
4. BIT2.update(r+1, r*v). The proper value from this range is $(r - l + 1) * v$. To cancel update 3 and give the proper value add $r * v$ because $r * v - (l - 1) * v = (r - l - 1) * v$.

To query $a[1] + ... + a[w]$: BIT1.query(w)*w + BIT2.query(w). We can initialize the BITs by using a size 1 range update for every initial value. This is still $O(n \log n)$ construction time.

3 Problems

1. You're given n $(1 \leq n \leq 10^5)$ horizontal line segments, each with inclusive endpoints (x_1, y) and (x_2, y) where $-10^9 \leq x_1 \leq x_2 \leq 10^9$. Each line segment has a value v $(-10^9 \leq v \leq 10^9)$.

   Answer each of q $(1 \leq q \leq 10^5)$ queries. Each query is of the form x', a, b, and asks you to sum the values of the a-th to the b-th (sorted by increasing y) line segments at the vertical line $x = x'$.
2. (Brian Dean, 2012) FJ 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 $L * C$. FJ notices several occurrences of one cow overtaking another. Count the total number of crossing events during the entire race.
3. (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.
4. (SPOJ BRCKTS) Given a bracket expression of length N $(1 \leq N \leq 30,000)$, process M operations. There are two types of operations, a replacement, which changes the i-th bracket into its opposite, and a check, which determines whether a bracket expression is correct.