

Binary Indexed Trees

Aarav Bajaj

December 2020

1 Introduction to Range Queries

Range Queries are a very common type of problem at the USACO Gold and Platinum divisions. They involve queries which ask to find the result of some operation applied to a specific range of elements in an array and sometimes involve updating certain elements in the array. To solve these problems efficiently, we use Data Structures. One of the most useful data structures to handle sum queries/updates is the Binary Indexed Tree (BIT).

Example:

You are given an array of length N ($1 \leq N \leq 100,000$) integers. Then, you are given Q ($1 \leq Q \leq 100,000$) queries. There are two types of queries:

1. Given two integers i and j , change the i th number in the array to j .
2. Given an integer l and r , output the sum of the numbers between the l th and r th numbers in the array.

The naive solution is to loop through all elements from position l to position r and compute for sum queries, and to simply change the value in the array for update queries. This results in an $O(N)$ runtime for sum and $O(1)$ runtime for update queries. The total time complexity will be $O(NQ)$ on average if we have many sum queries.

Another solution is to maintain a prefix-sum array - an array which stores the sum of the first i elements of the given array at position i - to answer sum queries in $O(1)$ time. To update the element at position i , we now need to update the prefix-sum array at every position after i , so we have an $O(N)$ time complexity for update queries. The total time complexity will be $O(NQ)$ on average if we have many update queries.

Unfortunately, none of these solutions are fast enough to solve the problem. A BIT will allow us to answer each query in $O(\log N)$ time, giving us a solution that will pass.

2 Introduction to Binary Indexed Trees

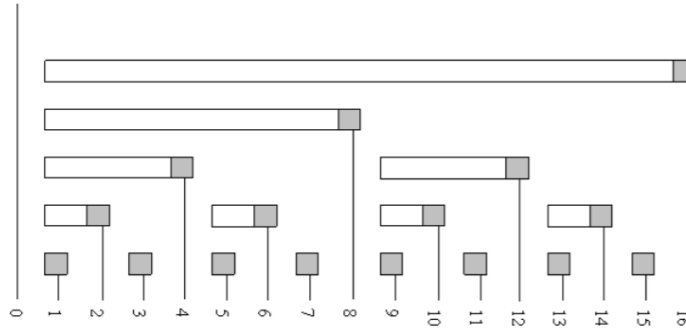
A Binary Index Tree is used for range sums and point updates (usually). Namely, a BIT can do element updates and prefix sums ($a[1] + a[2] + \dots + a[i]$; we one-index BITs for implementation-specific reasons) in $O(\log n)$. This is a tradeoff between a $O(n)$ update/ $O(1)$ query prefix-sum solution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

3 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 $a[1] + a[2] + \dots + a[14]$:



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

How do we find the prefix sum with this?

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

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

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.

3.1 Query

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

We use the two’s complement system. ($-1 = 1\dots1111_2$, $-2 = 1\dots1110_2$ 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: $-i = \sim(a1b)_2 + 1 = \sim a0\sim b + 1$. But b must consist of only zeros, since it’s after the lowest set bit. Therefore $\sim b + 1 = 100\dots$. Thus, we get $-i = (\sim a1b)_2$. 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;
}
```

3.2 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:

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

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 $O(\log n)$.

3.2.1 Range Update, Point Query

To increase an entire range of elements by some value v , we can simply update the ending index by v and the index before the starting index by $-v$. To query for a single element we only need to take the prefix-sum to that element (why?).

3.2.2 Range Update, Range Query

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 $a[1] + \dots + a[w]$: `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.

4 Counting Occurrences

One of the most useful ways to apply BITs involves counting the number of times something occurs in some range. The idea is to initialize all the values in a count array as 0, and maintain a BIT of it. Then every time something we want to count occurs, we update the value at the index of that location with a 1. If we take a range sum we can find the number of times something occurs in any range.

Example:

Given an array A of integers, an inversion is a pair of distinct indices (i, j) such that $A_i > A_j$ for $i < j$. Find the number of inversions in the array.

A naive solution would be to loop through all pairs in $O(n^2)$ time. We can optimize this by noticing that for each index, the number of inversions it forms with the elements before it is equal to the number of those elements that

are greater than it. We'll make a BIT that represents *the elements* of the array, instead of the index. If the BIT has a 1 at position i , then that means that we have seen the number i in the array. We'll start with a BIT of all zeroes and loop through from left to right. For the current element, we'll first do a sum query from that element to the end of the BIT. Then we'll take the value of the element and update that position in the BIT with a 1. Our answer is the sum of all of our sum queries. Since each query is $\log n$ with a BIT, we can solve the problem in $n \log n$ time.

Notice that we made use of the fact that at the time of processing each element, we had only processed the elements that occurred before it in the array, so they were the only elements that could possibly be filled with a 1. This, in combination with the point updates, allowed us to sum query for exactly we wanted.

5 Problems

1. USACO 2020 US Open Contest, Gold: Haircut
2. USACO 2017 January Contest, Gold: Balanced Photo
3. USACO 2017 February Contest, Gold: Why Did the Cow Cross the Road III
4. USACO 2019 January Contest, Gold: Sleepy Cow Sorting
5. USACO 2017 February Contest, Platinum: Why did the Cow Cross the Road
6. USACO 2015 February Contest, Gold: Cow Hopscotch
7. You are given an array of n ($1 \leq n \leq 2 \cdot 10^5$) integers and q ($1 \leq q \leq 2 \cdot 10^5$) queries of the form: how many distinct values are there in a range $[a, b]$?
8. 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 $x = x'$.
9. (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.
10. (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.
11. (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.

6 Extra

Here is a really nice tutorial if you are still unclear about how BIT works. It also has example code.

<https://www.hackerearth.com/practice/notes/binary-indexed-tree-or-fenwick-tree/>

This is a good explanation and list of different variants of the BIT. It has implementations in C++ as well.

https://cp-algorithms.com/data_structures/fenwick.html

This has a collection of good BIT resources and problem lists

<https://usaco.guide/gold/PURS>

7 References

- Patrick Zhang's 2019 BIT Lecture
- USACO Guide
- Competitive Programmer's Handbook