Range Queries

Kevin Shan*

December 2022

1 Introduction

Range Query data structures enable us to make range queries in logarithmic time, compared to the linear complexity of naive solutions. They also give point update or range update capabilities, in logarithmic time as well. Range Query data structures are extremely powerful and are commonly used in Gold and Platinum USACO problems.

2 Binary Indexed Trees Introduction

A Binary Indexed 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.

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 $a[1] + a[2] + ... + a[14]$:

$$14 \to 1110 \to 1100 \to 1000 \to 0$$

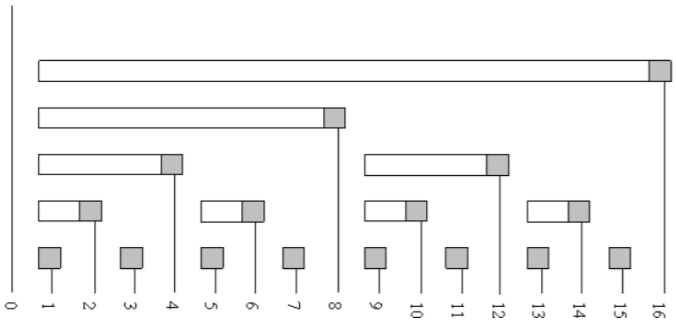
How do we find the prefix sum with this?

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

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

*Based on Charles Zhao's Segment Tree Lecture (2016) and Justin Zhang's Binary Indexed Trees Lecture (2017)

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?

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. 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);
}
```

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:

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

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 $O(\log n)$.

3.2.1 Range Updates

Range updates are a bit more involved, but can also be done in $O(\log n)$. The idea is to keep two 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 one BIT to calculate the $i * v$ parts and the other to calculate the $r * v$ and $(l-1) * v$ parts.

Specifically, here's how we'd update:

1. BIT1.update(l, v)
2. BIT1.update(r+1, -v)
3. BIT2.update(l, (l-1)*v)
4. BIT2.update(r+1, -r*v)

To query $a[1] + ... + a[w]$: BIT1.query(w)*w - BIT2.query(w). BIT1 handles cases where w falls within a range update; BIT2 handles the endpoints of the range updates.

4 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.

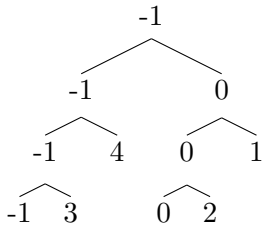
4. (SPOJ BRCKTS) Given a bracket expression of length N ($1 \leq N \leq 30,000$), process M operations. There are two types of operations, a replacement, which changes the i-th bracket into its opposite, and a check, which determines whether a bracket expression is correct.

5 Segment Trees Introduction

A *segment tree* is a data structure for storing intervals, or *segments*. Segment trees can efficiently answer dynamic range queries. We will use a segment tree to solve the Range Minimum Query (RMQ) problem, which is the problem of finding the minimum element in an array within a given range i to j. Other range queries include the maximum within a range or the sum of a range. A naive solution to RMQ is to iterate from index i to j, which takes $O(n)$ per query. This is too slow if n is large or if there are many queries. Another solution is to build a 2D matrix containing every single RMQ, which would be able to answer queries in $O(1)$ time. However, it would take $O(n^2)$ time to build this matrix and $O(n^2)$ space to store the matrix. Therefore, neither of these solutions scales well. Segment trees solve the problems of both time and space.

6 Constructing the Tree

A segment tree is a balanced binary tree in which each leaf represents an element in the array. The root of the tree represents segment $[0, n-1]$, and for each segment$[l, r]$ represented by the node at index p, the left child represents the segment $[l, (l+r)/2]$ and the right child represents the segment $[(l+r)/2+1, r]$. In the case of RMQ, "represents" means the value of the node is the minimum of the segment it represents. For example, for the array $[-1, 3, 4, 0, 2, 1]$, the tree would look as follows:



Constructing this tree takes $O(n)$ time and $O(n)$ space. In the pseudocode below, we build the tree recursively. The tree is represented as an array st where index 1 is the root of the tree and the left and right children of index p are indices $2 \times p$ and $(2 \times p) + 1$, respectively. l and r are the left and right bounds of the current segment, respectively.

Algorithm 1 Segment Tree Construction

```
function BUILD(p, l, r)
    if l = r then
        st[p] ← A[l]
    else
        pl ← 2 × p
        pr ← 2 × p + 1
        BUILD(pl, l, (l + r)/2)
        BUILD(pr, (l + r)/2, r)
        return MIN(st[pl], st[pr])
```

7 Solving Queries

There are three cases that we must consider when traversing a segment tree: when part of the segment represented by the node is within the query; when the segment is completely within the query; and when the segment is completely outside of the query. If part of the segment is within the query, we must check both of the node's children. If the segment is completely within the query, we return the node's value, which is the minimum of the segment it represents. If the segment is completely outside of the query, we return some very large number. In the pseudocode below, we traverse the tree recursively. With the segment tree built, solving an RMQ takes $O(\log n)$ time. This is because segment trees allow us to avoid traversing unrelated parts of the tree. In the worst case, in which only part of every segment we reach is within the query, we traverse two root-to-leaf paths, taking $O(2 \times \log n) = O(\log n)$ time.

Algorithm 2 Range Minimum Query Using a Segment Tree

```
function RMQ(p, l, r, i, j)
    if i > r or j < l then
        return ∞
    if l >= i & r <= j then
        return st[p]
    pl ← 2 × p
    pr ← 2 × p + 1
    minl ← RMQ(pl, l, (l + r)/2, i, j)
    minr ← RMQ(pr, (l + r)/2 + 1, r, i, j)
    return MIN(minl, minr)
```

8 Modifying the Tree

Remember that we said segment trees can efficiently answer *dynamic* range queries. This means that if the array on which we are performing RMQs changes, we can efficiently update the segment tree. If an element in the array changes, we start from the leaf node representing that element and move up the tree, updating nodes as we go. Thus, this takes $O(\log n)$ time.