Chapter –
Binary Index Trees

BINARY INDEXED TREE

Binary Indexed Tree also called Fenwick Tree provides a way to represent an array of numbers in an array, allowing prefix sums to be calculated efficiently.
Calculating prefix sums efficiently is useful in various scenarios. Let's start with a simple problem.

We are given an array a[], and we want to be able to perform two types of operations on it.

1. Change the value stored at an index i. (This is called a point update operation)
2. Find the sum of a prefix of length k. (This is called a range sum query)

A straightforward implementation of the above would look like this.

```
int a[] = {2, 1, 4, 6, -1, 5, -32, 0, 1};
void update(int i, int v)   //assigns value v to a[i]
{
    a[i] = v;
}
int prefixsum(int k)    //calculate the sum of all a[i] s
```

```
uch that 0 <= i < k
{
    int sum = 0;
    for(int i = 0; i < k; i++)
         sum += a[i];
    return sum;
}
```

This is a perfect solution, but unfortunately the time required to calculate a prefix sum is proportional to the length of the array, so this will usually time out when large number of such operations are performed.

Can We Do Better Than This?

One efficient solution is to use segment tree that can perform both operation in O(logN) time.
Using **Binary Indexed Tree** also, we can perform both the tasks in O(logN) time. But then why learn another data structure when segment tree can do the work for us. **It's because binary indexed trees require less space and are very easy to implement during programming contests (the total code is not more than 8-10 lines).**

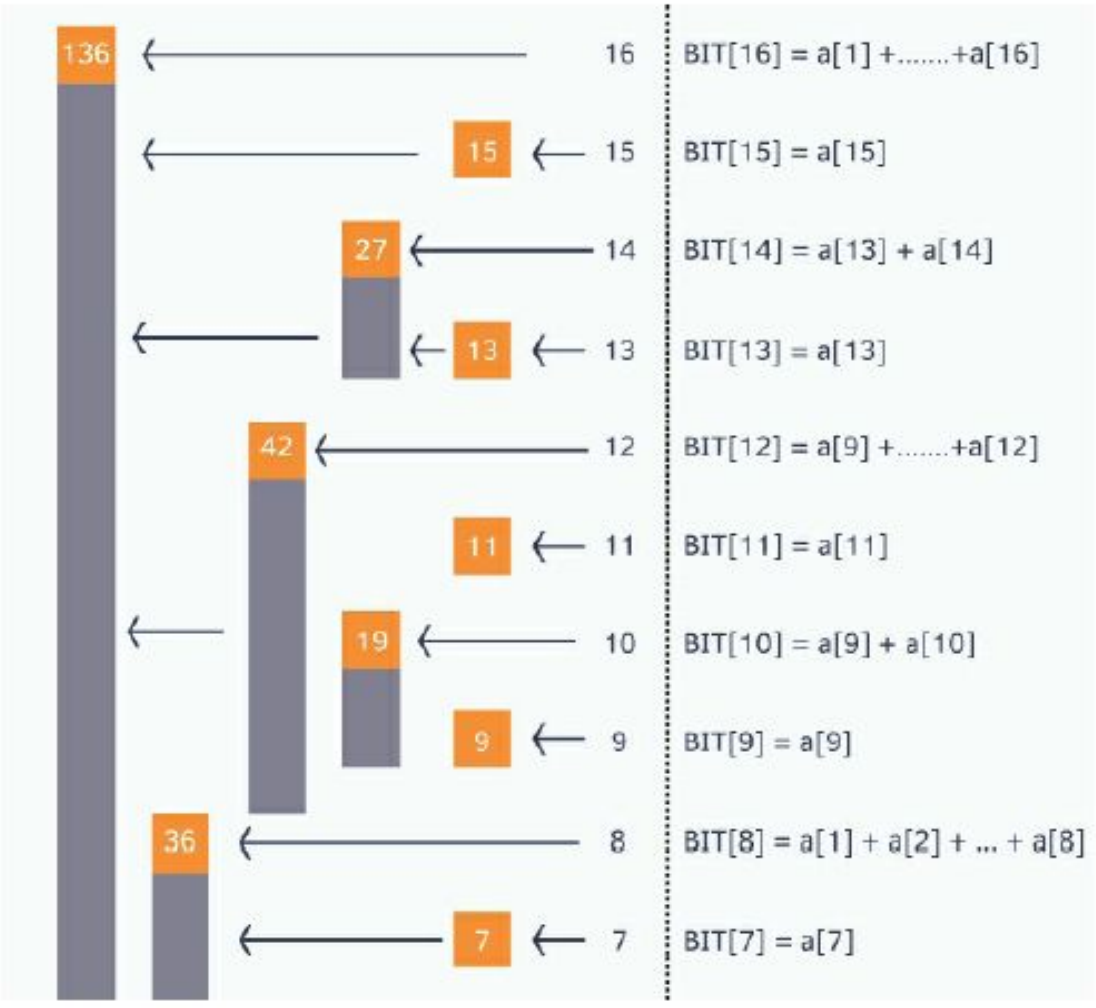
Basic Idea of Binary Indexed Tree :

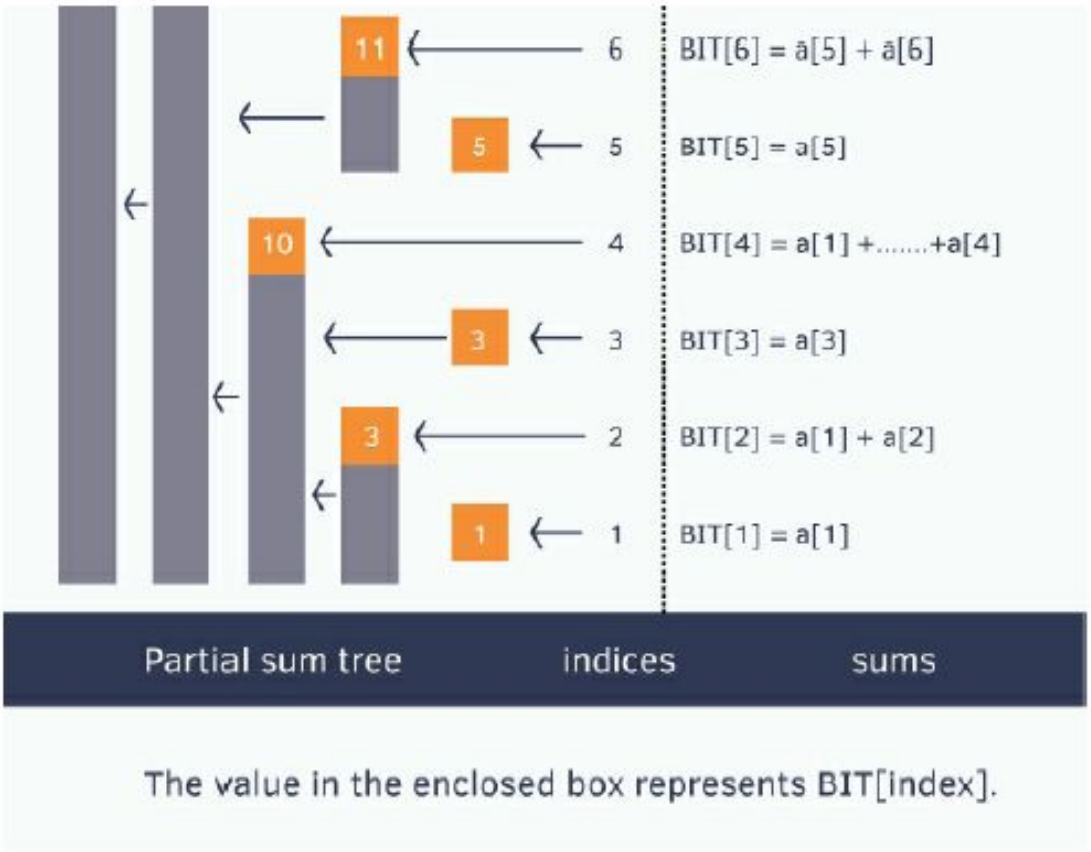
We know the fact that each integer can be represented as sum of

powers of two. Similarly, for a given array of size **N**, we can maintain an array **BIT[]** such that, at any index we can store sum of some numbers of the given array. This can also be called a partial sum tree.

Let's use an example to understand how **BIT[]** stores partial sums.

```
//for ease, we make sure our given array is 1-based index
ed
int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16};
```





The above picture shows the binary indexed tree, each enclosed box of which denotes the value BIT[index] and each BIT[index] stores partial sum of some numbers.

```
Every index i in the BIT[] array stores the cumulative su
m from the index i to i - (1<<r) + 1 (both inclusive), wh
ere r represents the last set bit in the index i.
```

Sum of first 12 numbers in array a[] = BIT[12] + BIT[8] = (a[12] + ... + a[9]) + (a[8] + ... + a[1])

Similarly, **sum of first 6 elements** = BIT[6] + BIT[4] = (a[6] + a[5]) + (a[4] + ... + a[1])

Sum of first 8 elements = BIT[8] = a[8] + ... + a[1]

Construct The BIT

Let's see how to construct this tree and then we will come back to querying the tree for prefix sums.

BIT[] is an array of size = 1 + the size of the given array a[] on which we need to perform operations. Initially all values in BIT[] are equal to 0. Then we call update() operation for each element of given array to construct the Binary Indexed Tree. The update() operation is discussed below.

```
void update(int index,int val) {
    while (index <= maxn) {
        tree[index] += val;
        index += (index & (-index));
    }
}
```

Suppose we call **update(13, 2).**

Here we see from the above figure that indices **13, 14, 16** cover index **13** and thus we need to add 2 to them also.

Initially x is 13, we update BIT[13]

```
BIT[13] += 2;
```

Now isolate the last set bit of x = 13(1101) and add that to x , i.e. **x += x&(-x)**

Last bit is of x = 13(1101) is 1 which we add to x, then x = 13+1 = 14, we update BIT[14]

```
BIT[14] += 2;
```

Now 14 is 1110, isolate last bit and add to 14, x becomes 14+2 = 16(10000), we update BIT[16]

```
BIT[16] += 2;
```

In this way, when an update() operation is performed on index x we update all the indices of BIT[] which cover index **x** and maintain the BIT[].

If we look at the for loop in update() operation, we can see that the loop runs at most the number of bits in index x which is restricted to be less or equal to n (the size of the given array), so we can say that the update operation takes at most **O(log2(n))** time.

Query in BIT

```
int query(int index) {
    int sum = 0;
    while (index > 0) {
        sum += tree[index];
        index -= (index & (-index));
    }
```

```
    return sum;
}
```

The above function query() returns the sum of first x elements in given array. Let's see how it works.

Suppose we call query(14), initially sum = 0

x is 14(1110) we add BIT[14] to our sum variable, thus sum = **BIT[14] = (a[14] + a[13])**

now we isolate the last set bit from x = 14(1110) and subtract it from x

last set bit in 14(1110) is 2(10), thus x = 14 – 2 = 12

we add BIT[12] to our sum variable, thus **sum = BIT[14] + BIT[12] = (a[14] + a[13]) + (a[12] + ... + a[9])**

again we isolate last set bit from x = 12(1100) and subtract it from x

last set bit in 12(1100) is 4(100), thus x = 12 – 4 = 8

we add BIT[8] to our sum variable, thus

sum = BIT[14] + BIT[2] + BIT[8] = (a[14] + a[13]) + (a[12] + ... + a[9]) + (a[8] + ... + a[1])

once again we isolate last set bit from x = 8(1000) and subtract it from x

last set bit in 8(1000) is 8(1000), thus x = 8 – 8 = 0

since x = 0, the for loop breaks and we return the prefix sum.

Talking about complexity, again we can see that the loop iterates at most the number of bits in x which will be at most n(the size of the given array). Thus the **query operation takes O(log2(n)) time**.

When To use BIT ?

Before going for Binary Indexed tree to perform operations over range, one must confirm that the operation or the function is:

Associative. i.e f(f(a, b), c) = f(a, f(b, c)) this is true even for seg-tree

Has an inverse. eg:

- addition has inverse subtraction (this example we have discussed)
- Multiplication has inverse division
- gcd() has no inverse, so we can't use BIT to calculate range gcd's

PROBLEM : INVCNT (Inversion Count)

Let **A[0...n - 1]** be an array of n distinct positive integers. If **i < j and A[i] > A[j]** then the pair **(i, j)** is called an **inversion of A**. Given n and an array A your task is to **find the number of inversions of A**.
For example, the array **a={2,3,1,5,4}** has three inversions: **(1,3), (2,3), (4,5), for the pairs of entries (2,1), (3,1), (5,4)**.

We'll use BIT to solve the problem of Inversion Count.

Replacing the Values of the Array with the Indexes

Usually when we are implementing a BIT is necessarily to map the original values of the array to a new range with values between [1,N], where N is the size of the array. This is due to the following reasons:

1. The values in one or more A[i] entry are too high or too low. (e.g. 10^12 or 10^−12).

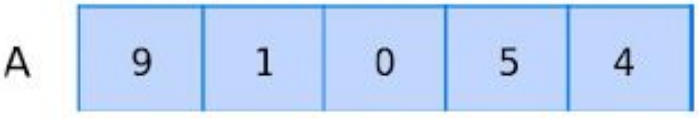
For example imagine that we are given an array of 3 integers: {1,10^12,5}
This means that if we want to construct a frequency table for our BIT data structure, we are going to need at least an array of 10^12 elements.

2. The values in one or more A[i] entry are negative.
   Because we are using arrays it's not possible to handle in our

BIT frequency of negative values (e.g. we are not able to do freq[−12]).

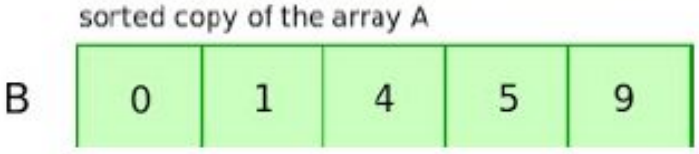
A simple way to deal with this issues is to replace the original values of the target array for indexes that maintain its relative order.

For example, given the following array:



The first step is to make a copy of the original array **A** let's call it **B**.

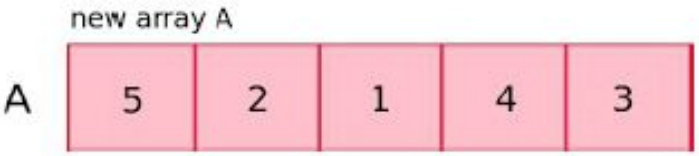
Then we proceed to sort **B** in non-descending order as follow:



Using binary search over the array B we are going to seek for every element in the array A, and stored the resulting position indexes (1-based) in the new array A.

binary_search(B,9)=4 found at position 4 of the array B
binary_search(B,1)=1 found at position 1 of the array B
binary_search(B,0)=0 found at position 0 of the array B
binary_search(B,5)=3 found at position 3 of the array B
binary_search(B,4)=2 found at position 2 of the array B

The resulting array after increment each position by one is the following:



The following C++ code fragment illustrate the ideas previously explained:

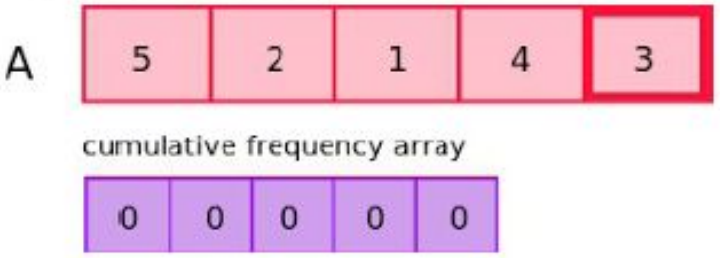
```
for(int i = 0; i < N; ++i)
   B[i] = A[i]; // copy the content of array A to array B

sort(B, B + N); // sort array B

for(int i = 0; i < N; ++i) {
   int ind = int(lower_bound(B, B + N, A[i]) - B);
   A[i] = ind + 1;
}
```

Counting inversions with the accumulate frequency

Initially the cumulative frequency table is empty, we start the process with the element **3**, the last one in our array.



how many numbers less than 3 have we seen so far

x=read(3−1) = 0

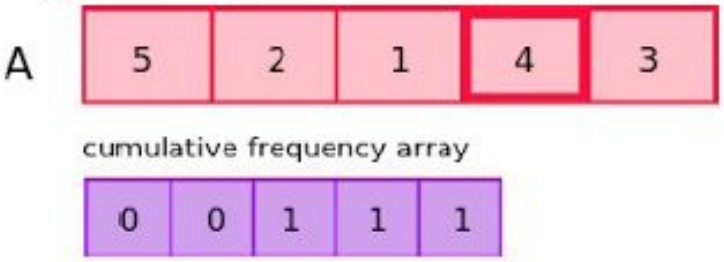
inv_counter=inv_counter+x

update the count of 3's so far

update(3,+1)

inv_counter=0

The cumulative frequency of value 3 was increased in the previous step, this is why the read(4−1) count the inversion (4,3).



how many numbers less than 4 have we seen so far

x=read(4−1)=1

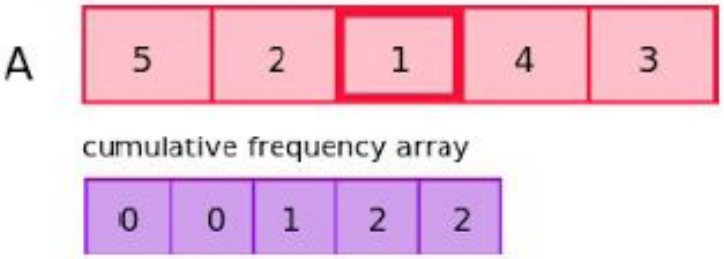
inv_counter=inv_counter+x

update the count of 4's so far

update(4,+1)

inv_counter=1

The term 1 is the lowest in our array, this is why there is no inversions beginning at 1.



how many numbers less than 1 have we seen so far

x=read(1−1)=0

inv_counter=inv_counter+x

update the count of 1's so far

update(1,+1)

inv_counter=1

And so on for the other elements...

PROBLEM : BAADSHAH

You are provided an array consisting of 'n' integers and asked to do following operations:

1. Update the position at "x" of array to "y" ie A[x]=y.
2. Find if there is any prefix in the array whose sum equals to "S".If yes, then output "Found" and the last index of prefix, else output "Not Found".

INPUT :

```
4 4
1 2 3 4
2 6
1 4 10
2 16
2 5
```

OUTPUT :

```
Found 3
```

```
Found 4
Not Found
```

https://www.codechef.com/problems/BAADSHAH

Construct a BIT(Binary Indexed Tree) from the given array, where internal nodes represents range sum. Use Point updates for query 1, and for query 2, using Binary search on range-query sums, where starting range is 1 to n, check if the prefix sum exists or not.

CODE :

```
ll read(ll index) {
    ll sum = 0;
    while (index > 0) {
        sum += BIT[index];
        index -= (index & (-index));
    }
    return sum;
}

void update(ll index, ll val) {
    while (index < maxn) {
        BIT[index] += val;
        index += (index & (-index));
    }
}
```

```
int main() {
    cin >> N >> M;
    for (int i = 1; i <= N; ++i) {
        cin >> arr[i];
        update(i, arr[i]);
    }

    while (M--) {
        ll idx;
        cin >> type;
        if (type == 1) {
            cin >> a >> b;
            update(a, b - arr[a]);
            arr[a] = b;
        }
        else {
            bool flag = false;
            cin >> a;
            ll lo = 1, hi = N;
            while (lo <= hi) {
                int mid = (lo + hi) >> 1;
                ll ans = read(mid);
                if (ans < a) {
                    lo = mid + 1;
                }
                else if (ans > a) {
                    hi = mid - 1;
                }
```

```
                else {
                    flag = true;
                    idx = mid;
                    break;
                }
            }
            if (flag)cout << "Found " << idx << "\n";
            else cout << "Not Found\n";
        }
    }
    return 0;
}
```

PROBLEM : CTRICK (CARD TRICK)

The magician shuffles a small pack of cards, holds it face down and performs the following procedure:
The top card is moved to the bottom of the pack. The new top card is dealt face up onto the table. It is the Ace of Spades.
Two cards are moved one at a time from the top to the bottom. The next card is dealt face up onto the table. It is the Two of Spades.
Three cards are moved one at a time...
This goes on until the nth and last card turns out to be the n of Spades.
This impressive trick works if the magician knows how to arrange the cards beforehand (and knows how to give a false shuffle). Your program has to determine the initial order of the cards for a given number of cards, $1 \leq n \leq 20000$.

INPUT :

```
2
4
5
```

OUTPUT :

```
2 1 4 3
3 1 4 5 2
```

We are asked to place 1 in 2nd free cell of our answer, then to place 2 in 3rd free cell of our answer while starting counting from position where we had placed 1 (and starting from the beginning if we reached end of array), then to place 3 in 4th free cell, and so on.

Now add one more optimization. At every moment we know how many free cells are in our array now. Assume we are at position X now, need to move 20 free cells forward, and you know that at given moment there are 2 free cells before X and 4 free cells after X. It means that we need to find 22nd free cell starting from beginning of array, which is same as going full circle 3 times and then taking 4th free cell. As we can see, we moved to a problem "find i-th zero in array".

CODE :

```
int read(int ind) {
    int sum = 0;
```

```
    while (ind > 0) {
        sum += tree[ind];
        ind -= (ind & -ind);
    }
    return sum;
}

void update(int ind, int val) {
    while (ind < 20002) {
        tree[ind] += val;
        ind += (ind & -ind);
    }
}
int ans[20005];
int b_search(int val) {

    int l = 1, r = n;
    while (l < r) {

        int mid = (l + r) >> 1;
        int tt = read(mid);

        if (tt > val) {
            r = mid;
        }
        else if (tt < val) {
            l = mid + 1;
        }
```

```
        else {
            r = mid;
            if (ans[mid] == 0) return mid;
        }
    }
    return l;

}
int main() {

    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    cin >> t;
    while (t--) {

        cin >> n;
        memset(tree, 0, sizeof(tree));
        for (int i = 1; i <= n;i++) {
            update(i, 1);
        }
        int fcells = n;
        int cur = 1;
        for (int i = 1; i < n + 1;++i) {
            int free = i + 1;
            int freeb = read(cur - 1);
            int freef = read(n) - freeb;
            int yy = free + freeb;
```

```
            int cell = yy % fcells;
            if (cell == 0) cell = fcells;
            int ret_ind = b_search(cell);
            ans[ret_ind] = i;
            fcells--;
            update(ret_ind, -1);
            cur = ret_ind;
        }

        for (int i = 1; i < n + 1;++i) {
            cout << ans[i] << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

TRY PROBEM : http://www.spoj.com/problems/DQUERY/

CODING BLOCKS
Code Your Way To Success

Created By *Shubham Rawat*