**How Does Binary Index Tree Work?**

The idea is based on the fact that all positive integers can be represented as sum of powers of 2. For example 19 can be represented as 16 + 2 + 1. Every node of BI Tree stores sum of n elements where n is a power of 2. For example, in the above first diagram for getSum(), sum of first 12 elements can be obtained by sum of last 4 elements (from 9 to 12) plus sum of 8 elements (from 1 to 8). The number of set bits in binary representation of a number n is O(Logn). Therefore, we traverse at-most O(Logn) nodes in both getSum() and update() operations. Time complexity of construction is O(nLogn) as it calls update() for all n elements.

Now, what does n&(-n) do?

Now, consider n=3.

The 8 bit signed binary representation of n=3 is: 00000011

The 8 bit singed representation of n=3 is: 11111101

Now, anding both gives you a number in which only last bit is set

N&(~N+1) would also gives the same. Since, ~N=-(N+1)

Now, the source code with command line is:

int get\_sum(int \*BIT,int i)

{

int index=i+1;

int val=0;

for(;index>0;)

{

val+=BIT[index];

index-=(index & -index);

}

**/\*for index 0, return BIT[1]**

**\*for index 0..1 return BIT[2]**

**\*for index 0..2 return BIT[3]+BIT[2]**

**\*for index 0..3 return BIT[4]**

**for index 0..4 return BIT[5]+BIT[4]**

**for index 0...5 return BIT[6]+BIT[4]**

**for index 0..6 rturn BIT[7]+BIT[6]+BIT[4]**

**for index 0..7 return BIT[8]**

**for index 0..8 return BIT[9]+BIT[8]**

**for index 0..9 contains BIT[10]+BIT[8]**

**for index 0...10 return BIT[11]+BIT[10]+BIT[8]**

**for index 0...11 return BIT[12]+BIT[8]**

**\*/**

return val;

}

void construct\_BIT\_util(int \*BIT,int index,int n,int val)

{

for(;index<=n;)

{

BIT[index]+=val;

index+=(index & (-index));

//incremented by the value in which only last bit of index is set

}

**//array indexing is in 0 indexing but**

**//for index 1, the element is added in 1,2,4,8**

**//for index 2, the element is added in 2,4,8**

**//for element 3, the element is added in 3,4,8**

**//for element 4, the element is added in 4,8**

**//for element 5, the element is added in 5,6,8**

**//for element 6, the element is added in 6,8**

**//for element 7, the element is added is added in 7,8**

**//for element 8, the element is added is added 8**

**//for element 9,the element is 9,10,12**

**//for element 10, the element is 10,12**

**//for element 11, the element is 11,12**

**//for element 12, the element is 12**

**/\***

**so, the element 1 contains only 1 i.e. arr[0]**

**the element 2 contains sum of element1+element2 i.e. arr[0..1]**

**the element 3 contains only 3 i.e. arr[2]**

**the element 4 contains arr[0...3]**

**the element 5 contains arr[4]**

**the element 6 contains arr[4..5]**

**the element 7 contains arr[6]**

**the element 8 contains arr[0..7]**

**the element 9 contains arr[8]**

**the element 10 contains arr[8..9]**

**the element 11 contains arr[10]**

**the element 12 contains arr[8..11]**

**\*/**

}

int \*constructBITree(int arr[], int n)

{

// Create and initialize BITree[] as 0

int \*BITree = new int[n+1];

//now, since, in BITree things starts from 1 indexing

//We need new int[n+1]

for (int i=1; i<=n; i++)

BITree[i] = 0;

// Store the actual values in BITree[] using update()

for (int i=0; i<n; i++)

construct\_BIT\_util(BITree, n, i, arr[i]);

// Uncomment below lines to see contents of BITree[]

//for (int i=1; i<=n; i++)

// cout << BITree[i] << " ";

return BITree;

}

int main()

{

int arr[]={2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9};

int n=sizeof(arr)/sizeof(arr[0]);

int \*BIT=construct\_BIT(arr,n);

printf("Sums of elements in arr[0..5] is %d\n",get\_sum(BIT,5));

arr[3]+=6;

update(BIT,n,3,6);

printf("sums of elements in arr[0..5] is %d\n",get\_sum(BIT,5));

return 0;

}

**Note:**

Now, another BIT specification is: BIT[6] will contain arr[0…5]

So, if you want to know the sum of sum arr[l…r]=BIT[r+1]-BIT[l]

**Binary Indexed Tree: Range Updates And Point Queries:**

**Problem statement 1:**

Given an array arr[0..n-1]. **Every element of array is initiated with 0. (This is a must condition)** The following operations need to be performed.

update(l, r, val) : Add ‘val’ to all the elements in the array from [l, r].

getElement(i) : Find element in the array indexed at ‘i’.

**Method 1 [update : O(n), getElement() : O(1)]**

**update(l, r, val) :** Iterate over the sub-array from l to r and increase all the elements by val.

**getElement(i) :** To get the element at ith index, simply return arr[i].

The time complexity in worst case is O(q\*n) where q is number of queries and n is number of elements.

This way is very easy to understand.

We can avoid updating all elements and can update only 2 indexes of the array!

**Method 2 [update : O(1), getElement() : O(n)]**

**update(l, r, val) :** Add ‘val’ to the lth element and subtract ‘val’ from the (r+1)th element, do this for all the update queries.

arr[l] = arr[l] + val

arr[r+1] = arr[r+1] - val

**getElement(i) :** To get ith element in the array find the sum of all integers in the array from 0 to i.(Prefix Sum).

Let’s analyze the update query. Why to add val to lth index? Adding val to lth index means that all the elements after l are increased by val, since we will be computing the prefix sum for every element. Why to subtract val from (r+1)th index? A range update was required from [l,r] but what we have updated is [l, n-1] so we need to remove val from all the elements after r i.e., subtract val from (r+1)th index. Thus the val is added to range [l,r]. (ALSO, We are calculating summation from 0 to ith index, to return getElement. I mean we are calculating prefix sum here)

// C++ program to demonstrate Range Update

// and Point Queries Without using BIT

#include <iostream>

using namespace std;

// Updates such that getElement() gets an increased

// value when queried from l to r.

void update(int arr[], int l, int r, int val)

{

arr[l] += val;

arr[r+1] -= val;

}

// Get the element indexed at i

int getElement(int arr[], int i)

{

// To get ith element sum of all the elements

// from 0 to i need to be computed

int res = 0;

for (int j = 0 ; j <= i; j++)

res += arr[j];

return res;

}

// Driver program to test above function

int main()

{

int arr[] = {0, 0, 0, 0, 0};

int n = sizeof(arr) / sizeof(arr[0]);

int l = 2, r = 4, val = 2;

update(arr, l, r, val);

//Find the element at Index 4

int index = 4;

cout << "Element at index " << index << " is " <<

getElement(arr, index) << endl;

l = 0, r = 3, val = 4;

update(arr,l,r,val);

//Find the element at Index 3

index = 3;

cout << "Element at index " << index << " is " <<

getElement(arr, index) << endl;

return 0;

}

**Note: consider the terminal case:**

int l = 2, r = 4, val = 2;

update(arr, l, r, val);

//Find the element at Index 4

int index = 4;

cout << "Element at index " << index << " is " <<

getElement(arr, index) << endl;

Now, here, the **arr[r+1]=-val;** is performed to a out of bound index integer. Since, memory allocation is static, it wont give any error. We should refrain from doing it.

**BIT based Method:**

In method 2, we have seen that the problem can reduced to update and prefix sum queries. We have seen that BIT can be used to do update and prefix sum queries in O(Logn) time.

Below is C++ implementation.

// C++ code to demonstrate Range Update and

// Point Queries on a Binary Index Tree

#include <iostream>

using namespace std;

// Updates a node in Binary Index Tree (BITree) at given index

// in BITree. The given value 'val' is added to BITree[i] and

// all of its ancestors in tree.

void updateBIT(int BITree[], int n, int index, int val)

{

// index in BITree[] is 1 more than the index in arr[]

index = index + 1;

// Traverse all ancestors and add 'val'

while (index <= n)

{

// Add 'val' to current node of BI Tree

BITree[index] += val;

// Update index to that of parent in update View

index += index & (-index);

}

}

// Constructs and returns a Binary Indexed Tree for given

// array of size n.

int \*constructBITree(int arr[], int n)

{

// Create and initialize BITree[] as 0

int \*BITree = new int[n+1];

for (int i=1; i<=n; i++)

BITree[i] = 0;

// Store the actual values in BITree[] using update()

for (int i=0; i<n; i++)

updateBIT(BITree, n, i, arr[i]);

// Uncomment below lines to see contents of BITree[]

//for (int i=1; i<=n; i++)

// cout << BITree[i] << " ";

return BITree;

}

// SERVES THE PURPOSE OF getElement()

// Returns sum of arr[0..index]. This function assumes

// that the array is pre processed and partial sums of

// array elements are stored in BITree[]

int getSum(int BITree[], int index)

{

int sum = 0; // Initialize result

// index in BITree[] is 1 more than the index in arr[]

index = index + 1;

// Traverse ancestors of BITree[index]

while (index>0)

{

// Add current element of BITree to sum

sum += BITree[index];

// Move index to parent node in getSum View

index -= index & (-index);

}

return sum;

}

// Updates such that getElement() gets an increased

// value when queried from l to r.

void update(int BITree[], int l, int r, int n, int val)

{

// Increase value at 'l' by 'val'

updateBIT(BITree, n, l, val);

// Decrease value at 'r+1' by 'val'

updateBIT(BITree, n, r+1, -val);

}

// Driver program to test above function

int main()

{

int arr[] = {0, 0, 0, 0, 0};

int n = sizeof(arr)/sizeof(arr[0]);

int \*BITree = constructBITree(arr, n);

// Add 2 to all the element from [2,4]

int l = 2, r = 4, val = 2;

update(BITree, l, r, n, val);

// Find the element at Index 4

int index = 4;

cout << "Element at index " << index << " is " <<

getSum(BITree,index) << "\n";

// Add 2 to all the element from [0,3]

l = 0, r = 3, val = 4;

update(BITree, l, r, n, val);

// Find the element at Index 3

index = 3;

cout << "Element at index " << index << " is " <<

getSum(BITree,index) << "\n" ;

return 0;

}

Element at index 4 is 2

Element at index 3 is 6

**Now, remember the given definition of getElement(i)**

**getElement(i) :** To get ith element in the array find the sum of all integers in the array from 0 to i.(Prefix Sum).

That is why getSum(BITree,index) satisfies getElement(i).

**Time Complexity :** O(q \* log n) + O(n \* log n) where q is number of queries.

Now, notice that, for first two methods:

The time complexity is O(q.n) where q is number of queries and n is number of elements.

**Binary Indexed Tree based method:** update arr[l]…to arr[n-1] by val (updateBit works in same way)

Then update arr[r+1]…to arr[n-1] by = -val (updateBit works in same way)

Now, finally return arr[0..n-1] as by returning BIT[n]

**Problem Statement 2:**

Given an array arr[0..n-1]. **It must be initialized to 0. (must)** The following operations need to be performed.

update(l, r, val) : Add ‘val’ to all the elements in the array from [l, r].

getRangeSum(l, r) : Find sum of all elements in array from [l, r].

Initially all the elements in the array are 0. Queries can be in any order, i.e., there can be many updates before range sum.

**Solution:**

An Efficient Solution is to make sure that both queries can be done in O(Log n) time. We get range sum using prefix sums. How to make sure that update is done in a way so that prefix sum can be done quickly? Consider a situation where prefix sum [0, k] (where 0 <= k < n) is needed after range update on range [l, r]. Three cases arises as k can possibly lie in 3 regions.

Case 1: 0 < k < l

The update query won’t affect sum query.

Case 2: l <= k <= r

Consider an example:

Add 2 to range [2, 4], the resultant array would be:

0 0 2 2 2

If k = 3

Sum from [0, k] = 4

How to get this result?

Simply add the val from lth index to kth index. Sum is incremented by “val\*(k) – val\*(l-1)” after update query.

Case 3: k > r

For this case, we need to add “val” from lth index to rth index. Sum is incremented by “val\*r – val\*(l-1)” due to update query.

Observations :

Case 1: is simple as sum would remain same as it was before update.

Case 2: Sum was incremented by val\*k – val\*(l-1). We can find “val”, it is similar to finding the ith element in range update and point query article. So we maintain one BIT for Range Update and Point Queries, this BIT will be helpful in finding the value at kth index. Now val \* k is computed, how to handle extra term val\*(l-1)?

In order to handle this extra term, we maintain another BIT (BIT2). Update val \* (l-1) at lth index, so when getSum query is performed on BIT2 will give result as val\*(l-1).

Case 3 : The sum in case 3 was incremented by “val\*r – val \*(l-1)”, the value of this term can be obtained using BIT2. Instead of adding, we subtract “val\*(l-1) – val\*r” as we can get this value from BIT2 by adding val\*(l-1) as we did in case 2 and subtracting val\*r in every update operation.

Update Query

Update(BITree1, l, val)

Update(BITree1, r+1, -val)

UpdateBIT2(BITree2, l, val\*(l-1))

UpdateBIT2(BITree2, r+1, -val\*r)

Range Sum

**getSum(BITTree1, k) \*k) - getSum(BITTree2, k).**

**Inversion Count:**

**// C++ program to count inversions using Binary Indexed Tree**

#include<bits/stdc++.h>

using namespace std;

// Returns sum of arr[0..index]. This function assumes

// that the array is preprocessed and partial sums of

// array elements are stored in BITree[].

int getSum(int BITree[], int index)

{

int sum = 0; // Initialize result

// Traverse ancestors of BITree[index]

while (index > 0)

{

// Add current element of BITree to sum

sum += BITree[index];

// Move index to parent node in getSum View

index -= index & (-index);

}

return sum;

}

// Updates a node in Binary Index Tree (BITree) at given index

// in BITree. The given value 'val' is added to BITree[i] and

// all of its ancestors in tree.

void updateBIT(int BITree[], int n, int index, int val)

{

// Traverse all ancestors and add 'val'

while (index <= n)

{

// Add 'val' to current node of BI Tree

BITree[index] += val;

// Update index to that of parent in update View

index += index & (-index);

}

}

// Converts an array to an array with values from 1 to n

// and relative order of smaller and greater elements remains

// same. For example, {7, -90, 100, 1} is converted to

// {3, 1, 4 ,2 }

void convert(int arr[], int n)

{

// Create a copy of arrp[] in temp and sort the temp array

// in increasing order

int temp[n];

for (int i=0; i<n; i++)

temp[i] = arr[i];

sort(temp, temp+n);

// Traverse all array elements

for (int i=0; i<n; i++)

{

// lower\_bound() Returns pointer to the first element

// greater than or equal to arr[i]

arr[i] = lower\_bound(temp, temp+n, arr[i]) - temp + 1;

}

}

// Returns inversion count arr[0..n-1]

int getInvCount(int arr[], int n)

{

int invcount = 0; // Initialize result

// Convert arr[] to an array with values from 1 to n and

// relative order of smaller and greater elements remains

// same. For example, {7, -90, 100, 1} is converted to

// {3, 1, 4 ,2 }

convert(arr, n);

// Create a BIT with size equal to maxElement+1 (Extra

// one is used so that elements can be directly be

// used as index)

int BIT[n+1];

for (int i=1; i<=n; i++)

BIT[i] = 0;

// Traverse all elements from right.

for (int i=n-1; i>=0; i--)

{

// Get count of elements smaller than arr[i]

invcount += getSum(BIT, arr[i]-1);

//here, marks are left by higher index and which are smaller than arr[i]

**//since, we can use element value as index getSum(BIT,arr[I]-1) will give count of numbers which are present from 0 to arr[l]-1**

**// Add current element to BIT**

**//mark the value**

updateBIT(BIT, n, arr[i], 1);

}

return invcount;

}

// Driver program

int main()

{

int arr[] = {8, 4, 2, 1};

int n = sizeof(arr)/sizeof(int);

cout << "Number of inversions are : " << getInvCount(arr,n);

return 0;

}

**Find Inversion Of Size 3:**

We can reduce the complexity if we consider every element arr[i] as middle element of inversion, find all the numbers greater than a[i] whose index is less than i, find all the numbers which are smaller than a[i] and index is more than i. We multiply the number of elements greater than a[i] to the number of elements smaller than a[i] and add it to the result.

**Now, how to implement it using BIT:**

To find out the number of smaller elements for an index we iterate from n-1 to 0. For every element a[i] we calculate the getSum() function for (a[i]-1) which gives the number of elements till a[i]-1.

To find out the number of greater elements for an index we iterate from 0 to n-1. For every element a[i] we calculate the sum of numbers till a[i] (sum smaller or equal to a[i]) by getSum() and subtract it from i (as i is the total number of element till that point) so that we can get number of elements greater than a[i].

// C++ program to count inversions of size three using

// Binary Indexed Tree

#include<bits/stdc++.h>

using namespace std;

// Returns sum of arr[0..index]. This function assumes

// that the array is preprocessed and partial sums of

// array elements are stored in BITree[].

int getSum(int BITree[], int index)

{

int sum = 0; // Initialize result

// Traverse ancestors of BITree[index]

while (index > 0)

{

// Add current element of BITree to sum

sum += BITree[index];

// Move index to parent node in getSum View

index -= index & (-index);

}

return sum;

}

// Updates a node in Binary Index Tree (BITree) at given index

// in BITree. The given value 'val' is added to BITree[i] and

// all of its ancestors in tree.

void updateBIT(int BITree[], int n, int index, int val)

{

// Traverse all ancestors and add 'val'

while (index <= n)

{

// Add 'val' to current node of BI Tree

BITree[index] += val;

// Update index to that of parent in update View

index += index & (-index);

}

}

// Converts an array to an array with values from 1 to n

// and relative order of smaller and greater elements remains

// same. For example, {7, -90, 100, 1} is converted to

// {3, 1, 4 ,2 }

void convert(int arr[], int n)

{

// Create a copy of arrp[] in temp and sort the temp array

// in increasing order

int temp[n];

for (int i=0; i<n; i++)

temp[i] = arr[i];

sort(temp, temp+n);

// Traverse all array elements

for (int i=0; i<n; i++)

{

// lower\_bound() Returns pointer to the first element

// greater than or equal to arr[i]

arr[i] = lower\_bound(temp, temp+n, arr[i]) - temp + 1;

}

}

// Returns count of inversions of size three

int getInvCount(int arr[], int n)

{

// Convert arr[] to an array with values from 1 to n and

// relative order of smaller and greater elements remains

// same. For example, {7, -90, 100, 1} is converted to

// {3, 1, 4 ,2 }

convert(arr, n);

// Create and initialize smaller and greater arrays

int greater1[n], smaller1[n];

for (int i=0; i<n; i++)

greater1[i] = smaller1[i] = 0;

// Create and initialize an array to store Binary

// Indexed Tree

int BIT[n+1];

for (int i=1; i<=n; i++)

BIT[i]=0;

for(int i=n-1; i>=0; i--)

{

smaller1[i] = getSum(BIT, arr[i]-1);

updateBIT(BIT, n, arr[i], 1);

}

// Reset BIT

for (int i=1; i<=n; i++)

BIT[i] = 0;

// Count greater elements

for (int i=0; i<n; i++)

{

greater1[i] = i - getSum(BIT,arr[i]);

updateBIT(BIT, n, arr[i], 1);

}

// Compute Inversion count using smaller[] and

// greater[].

int invcount = 0;

for (int i=0; i<n; i++)

invcount += smaller1[i]\*greater1[i];

return invcount;

}

// Driver program to test above function

int main()

{

int arr[] = {8, 4, 2, 1};

int n = sizeof(arr)/sizeof(arr[0]);

cout << "Inversion Count : " << getInvCount(arr, n);

return 0;

}