Fenwick Tree

It is a data structure that supports 2 kind of operations:

1. Point update: update(i, x) will increment a(i) by x.
2. Range query: query(i) will return the value of a(1) + a(2) + … + a(i)

Using array:
1. Point update can be done in O(1)
2. Range query can be done in O(n)

By maintaining a prefix array:
1. Point update can be done in O(n)
2. Range query can be done in O(1)

Fenwick tree allows us to perform both the operations in O(log(n)) by solving the shortcomings of these 2 approaches.

What are the shortcomings?

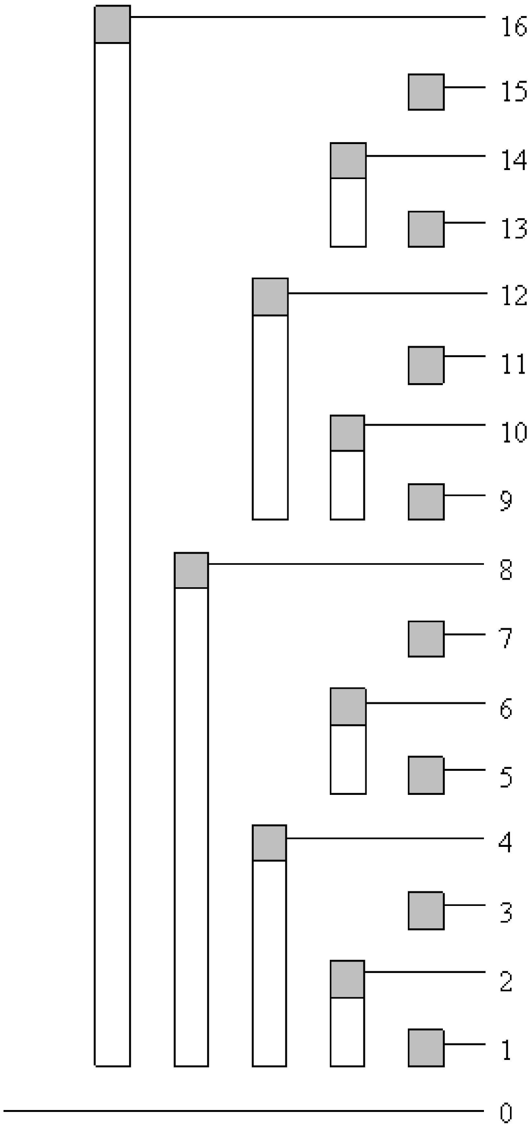
1. In a normal array, every index stores only the value of itself. So, to answer the query, there is no other way but to iterate through the range [l, r].

2. In the prefix array, every index i stores the sum of elements with index <= i. So, a(i) contributes its value to every index >= i in the prefix array.

3. In order to carry out both operations in O(log(n)), we must ensure that:

    a. t(i) contains a sum of a segment (g(i), i]. Also the range (0, i] should be expressible as a concatenation of at most log(n) such segments.

    b. a(i) should contribute its value to at most log(n) indexes of t. In another words, there should exists at most log(n) values of j satisfying:
    $$g(j) < i <= j <= n$$



Tree of responsibility
The segments represent the range whose sum is stored in t(i)

For example:

1. (0, 10] can be expressed as (0, 8] + (8, 10] . So, query(10) can be executed in 2 steps.

2. Index 10 contributes its value to segments (8, 10] , (8, 12] , (0, 16]. So, update(10, x) can be executed in 3 steps.

Definition of g(i)

Let g be a function defined in the domain of natural numbers. We first express i in its binary form, and then set the last 'on' bit to 'off'.

E.g:

10	=	1 0 1 0
g(10)	=	1 0 0 0
12	=	1 1 0 0
g(12)	=	1 0 0 0

So, g(10) = g(12) = 8

How to calculate g(i)

It turns out that $g(i) = i - (i \& -i)$

The contribution of the last set bit is equal to i & (-i).

E.g:

20	=	1 0 1 0 0
1's complement	=	0 1 0 1 1
2's complement	=	0 1 1 0 0
20 & (-20)	=	0 0 1 0 0

In a k-bit number, there can exist at most k 'on' bits. Hence we need to go through at most k segments to answer the query for any number < 2^k. It proves that the complexity of each query is O(log n).

How to update?

Whenever we add x to a(i), we must add x to all such t(j), where i lies inside the segment of t(j).

In other words, for all j satisfying $g(j) < i <= j <= n$, $t(j)$ must be incremented by x.

How to find such j ?

Let’s suppose there are total k ‘on’ bits in j.

Observation 1: The first k-1 ‘on’ bits of i and j must be in the same position.

Observation 2: The k^{th} ‘on’ bit of j should make sure that j >= i is satisfied.

How to iterate over such j ?

Let’s try to understand it using an example.

Let i = 1 1 0 0 1 0 1 0 1

For k = 5,
j = 1 1 0 0 1 0 1 0 1
j = 1 1 0 0 1 0 1 1 0

For k = 4,
j = 1 1 0 0 1 1 0 0 0

For k = 3,
j = 1 1 0 1 0 0 0 0 0
j = 1 1 1 0 0 0 0 0 0

For k = 2,
No such j

For k = 1,
j = 1 0 0 0 0 0 0 0 0 0
j = 1 0 0 0 0 0 0 0 0 0 0 … and so on

Again, it turns out that we can make transition from upper j to the lower j using a simple formula: $j = j + (j \& -j)$

Since the position of the last ‘on’ bit is decreasing at least by 1 in every transition, the complexity of update(i, x) is O(log n).

Implementation (One based Indexing):

```
struct BIT {

      int n;
      vector<int> t;

      BIT() {}
      BIT(int n) : n(n + 5), t(n + 5) {}

      // a[i] += val
      void update(int i, int val){
            for(; i < n; i += i & -i){
                  t[i] += val;
            }
      }

      // returns a[1] + a[2] + ... + a[i]
      int query(int i){
            int ret = 0;
            for(; i; i -= i & -i){
                  ret += t[i];
            }
            return ret;
      }

      // returns a[l] + a[l+1] + ... + a[r]
      int query(int l, int r){
            return query(r) - query(l - 1);
      }
};
```

Implementation (Zero based Indexing):

1. Instead of update(i, x), we execute update(i + 1, x)
2. Instead of query(i), we execute query(i + 1)

```
struct BIT {

      int n;
      vector<int> t;

      BIT() {}
      BIT(int n) : n(n + 5), t(n + 5) {}

      // a[i] += val
      void update(int i, int val){
            for(++i; i < n; i += i & -i){
                  t[i] += val;
            }
      }

      // returns a[0] + a[1] + ... + a[i]
      int query(int i){
            int ret = 0;
            for(++i; i; i -= i & -i){
                  ret += t[i];
            }
            return ret;
      }

      // returns a[l] + a[l+1] + ... + a[r]
      int query(int l, int r){
            return query(r) - query(l - 1);
      }
};
```