# **A Visual Introduction to Fenwick Tree**

<https://medium.com/carpanese/a-visual-introduction-to-fenwick-tree-89b82cac5b3c>

The Fenwick Tree, also called Binary Indexed Tree, is a data structure used to update elements and evaluate range queries in arrays. It differs from other similar data structures by using a method to arrange binary numbers in two dimensions.

Its straightforward implementation hides beautiful properties that explain how and why it works. This article tries to visually explain what a Fenwick Tree is, why it has to be 1-indexed, how to walk on the tree using the least significant bit, and, most importantly, how someone could come up with the idea of this structure, whose properties seem like magic.

# **The range sum query problem**

Before talking about the data structure itself, let us understand what problem it tries to solve.

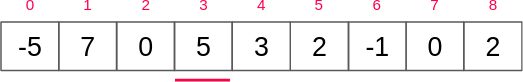
Given an array *v*, we have to evaluate two operations:

1. ****query(l, r):**** Find the sum of all elements in a given interval [*l, r*].
2. ****update(i, val):**** Change the value of position *i* of the array to *val*.

For example, for the array [-5, 7, 0, 1, 3, 2, -1, 0, 2], the result of query(1, 4) should return 11. However, if we call update(3, 5) and then check the value of query(1, 4), the result will be 15, not 11, because the array is different now. See figure 1.



****Figure 1a:**** query(1, 4) is 7+0+1+3=11.



****Figure 1b:****The value of position 3 becomes 5.



****Figure 1c:****query(1, 4) is now 7+0+5+3=15, not 11.

## **A naive solution**

To answer a query, just iterate through *v* from position *l* to position *r*, accumulating the sum of all elements. For an update, just change the value of the *i*-th position of *v*.

int query(int v[], int l, int r) {

int s = 0;

for (int i = l; i <= r; i++) {

s += v[i];

}

return s;

}

void update(int v[], int i, int val) {

v[i] = val;

}

**Code 1:** The naive solution.

This strategy allows an update in *O(1)* time complexity at the cost of having a query in *O(|v|)*.

## **Another naive solution**

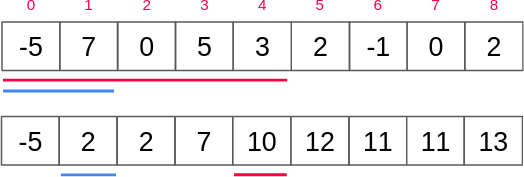
It is possible to invert the logic behind the previous solution.

It’ll use prefix sum of the array. The prefix sum *s*[*i*] of an array *v* is equal to *v*[*0*] + *v*[*1*] + … + *v*[*i*].



****Figure 2a:**** The building of the prefix sum of a given array. The upper array is the array v. The bottom array is the prefix sum s of v.

To answer a query, we return the difference between the *r*-th position and the (*l*-1)-th position. If *l=0*, then we return the *r*-th position. It can be done in O(1) time complexity.



****Figure 2b:**** We answer query(2, 4) by evaluating s[4] - s[1].

On the other hand, for each update, we have, in the worst scenario, to change the value of the first element, rebuilding the whole prefix sum array in *O(|v|)* time complexity.

int query(int s[], int l, int r) {

if (l == 0) return s[r];

else return s[r] - s[l - 1];

}

void update(int s[], int v[], int n, int i, int val) {

for (int j = i; j < n; j++) {

s[j] -= v[i];

s[j] += val;

}

v[i] = val;

}

**Code 2:** Another naive solution.

The first solution has a good update but a terrible query. The second one has a good query but a terrible update. The natural thing to do now is to ask if there is a solution such that both query and update are not so fast and not so slow.

## **Exercises**

****1.**** Instead of the sum, we want the minimum element in the range [*l*, *r*] of the array. Can we still use the second algorithm to do it? What do we have to change to have a query in O(1) time complexity?

# **The Fenwick Tree**

The Fenwick Tree improves the performance of the prefix sum, but it does not exactly solve the range sum query problem. It has some limitations. This structure can evaluate two different operations:

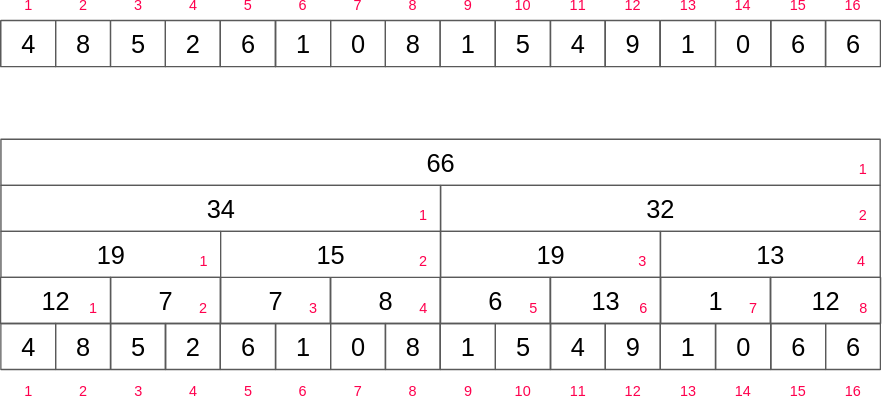
1. ****query(n):**** Find the sum of the first *n* elements.
2. ****add(i, val):**** Add *val* to the value of position *i* of the array.

It is easy to see how we can transform those operations into the ones we want.

## **The conception of the tree**

Let us build an intermediary binary tree such that its nodes contain the answer for a segment of the array. The root node, by definition, has the sum of the entire array. Its left child has the sum of the first half and the right child of the second half.

Generically, each node represents one segment of the array, its left child represents the first half of that segment, and the right child represents the second half.



****Figure 3:****For those familiar with it, that is precisely the segment tree of the array.

It will be clear later why it should be 1-indexed. It will also be clear what happens if the array is not a power of 2. Do not worry about that for now.

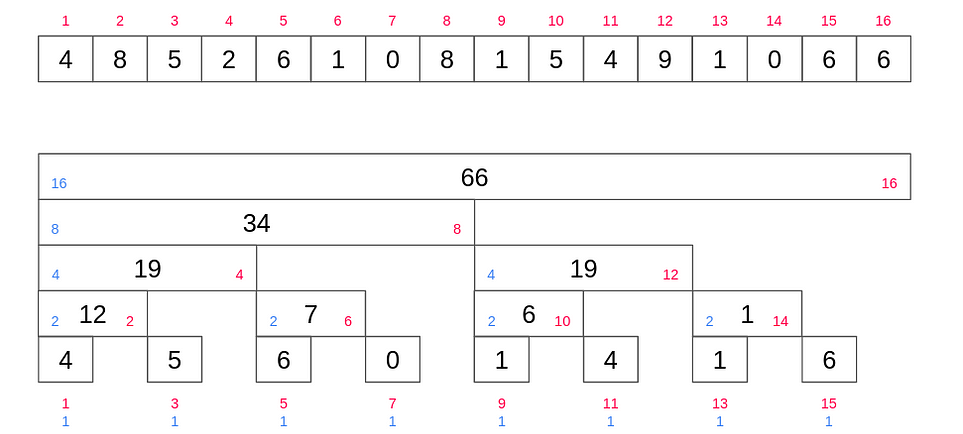
How can we use that new structure to sum the first *n* elements?



**This is a GIF**

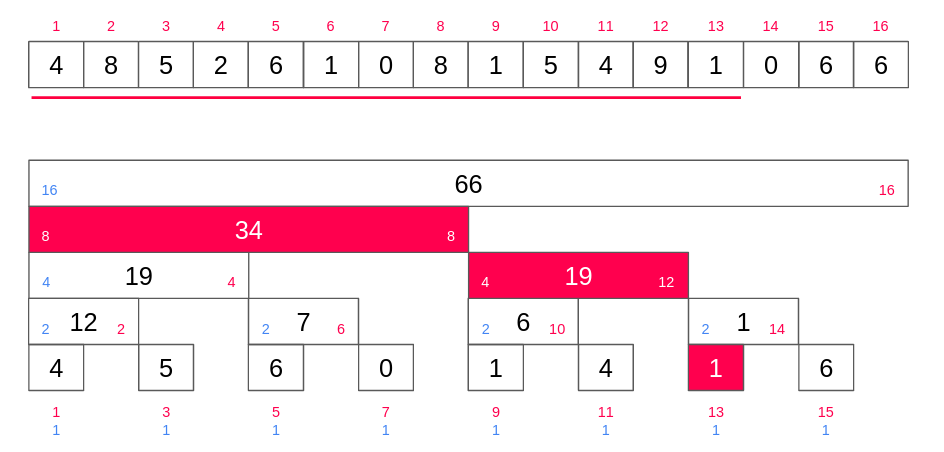
****Animation 1:****The answers for each one of query(i) for 1 ≤ i ≤ n. We achieve the answer manually, trying to figure out a pattern for an algorithm.

Let us call each element of that structure a block. Note that we can completely ignore the even indexes. That is because the block immediately above can express the same amount with one block less.



****Figure 4:****The summation always starting at the first node allows us to modify the segment tree to a new tree with unique properties. That new tree is called Fenwick Tree.

To understand what that new tree is doing, use index 13 as an example. 13 = (1101)₂ = 8+4+0+1. This structure divides the sum of the first 13 elements into three blocks, the first block of length 8, the second block of length 4, and the third block of length 1. See figure 5.

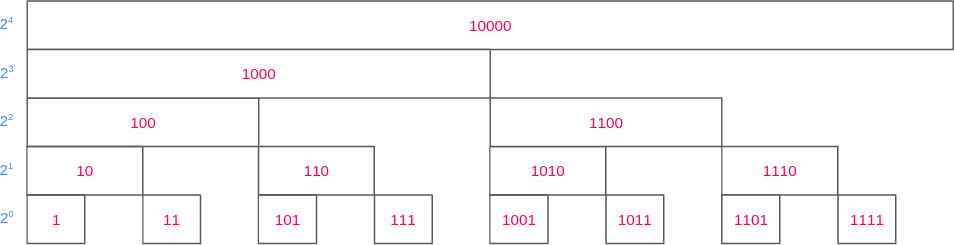


****Figure 5:****The frame of animation 1 for query(13).

*Generally, each set bit will be responsible for one part of the summation.*

In our example, to sum up the first 13 = (1101)₂ elements, we sum the values on position 13 = (1101)₂, 12 = (1100)₂, and 8= (1000)₂.

Although hard to see, that property is a straight consequence of how we construct the structure. Ignoring the even indexes on animation 1 is equivalent to a [carry-over](https://en.wikipedia.org/wiki/Carry_(arithmetic)" \t "https://medium.com/carpanese/_blank) on the binary representation of an array index.



****Figure 6:****A cleaner way to look at the construction we built. If we forget about the range sum query, we can use this structure for writing binary numbers.

We can now write a simple algorithm to evaluate the query operation.

#define lsb(n) (n & (-n))

int op(int a, int b) {

return a + b;

}

int query(int n) {

int res = 0;

while (n > 0) {

res = op(res, fenwick[n]);

n -= lsb(n);

}

return res;

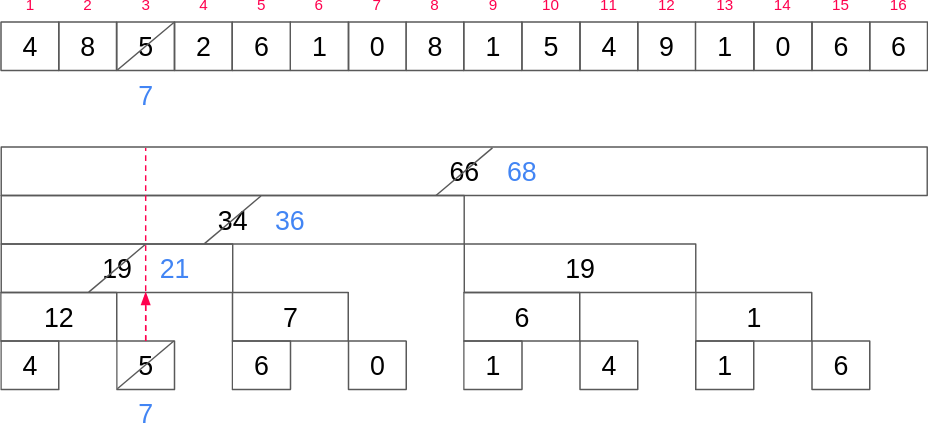
}

**Code 3:** The Fenwick Tree query C++ implementation. lsb macro evaluates the [least significant bit](https://en.wikipedia.org/wiki/Bit_numbering" \l "Least_significant_bit" \t "https://medium.com/carpanese/_blank) of a given number.

This data structure is called Fenwick Tree, or Binary Indexed Tree.

The update operation is a little trickier to implement, but it is easier to understand. All nodes “above” a leaf node contains the value of the leaf node in its sum. When we update a value, we start changing the value of the leaf node and updates all nodes “above” it.

Note that node A being an ancestor of node B does not necessarily mean that A is “above” B.



****Figure 7:****The update operation visualized.

To define those concepts better, we will need to understand how we can walk on the tree.

Nothing remarkably different happens if the array is not a power of 2. We used that property merely to make it easier to understand the intermediary structure. The unique assumption we need indeed is that the indexes can be expressed in binary form.



**This is a GIF**

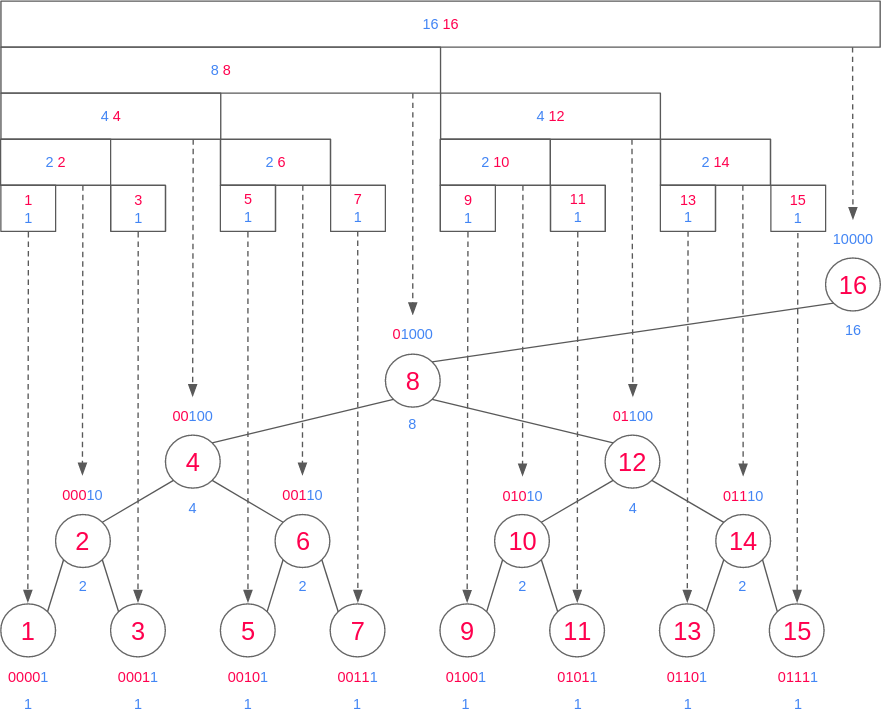
****Animation 2:****The answers for each one of query(i) for 1 ≤ i ≤ 10. It is identical to animation 1, but stopping earlier.

## **Walking on the tree**

Let us redraw the blocks to make a traditional tree-shape structure. Each block will be represented by a node.

From now on, we can ignore the content of the array and focus entirely on its indexes. Each index will be decomposed into two segments, the *prefix,*and *suffix*. The least significant bit will split those segments, and it will belong to the suffix.

*The prefix reveals the path to arrive at the node from the root. The suffix reveals the size of the block, the size of its prefix sum portion.* See figure 8.

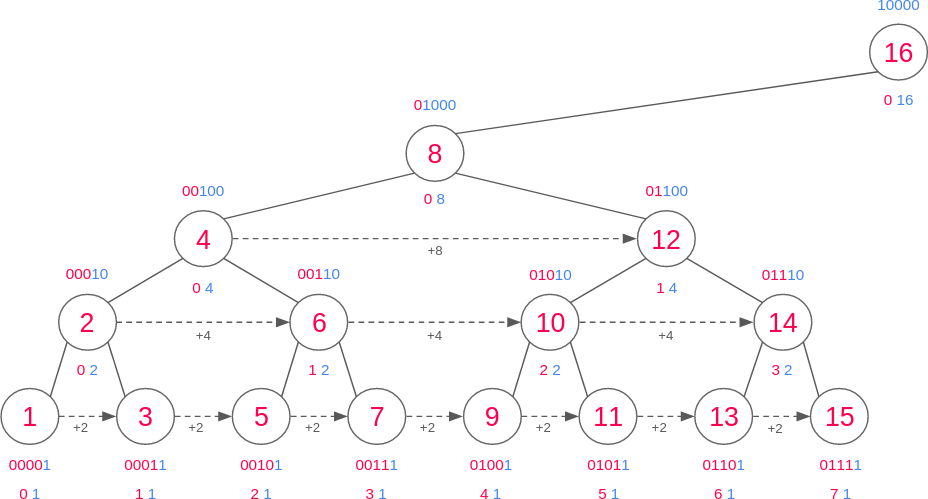


****Figure 8:****The Fenwick Tree.

By construction, all nodes have the same suffix at the same level of the tree. Consequently, the index of each node differs by a difference of a power of 2. See figure 9.

It is easy to understand why the suffix represents the size of the block and why all nodes in the same level have the same suffix. That property is a direct consequence of how we build the tree.

*That is why the Fenwick Tree must be 1-indexed.* This construction works if and only if there is at least one set bit. Zero has no least significant bit, which means that the size of its respective block is zero. A block with no size does not make any sense in our context.

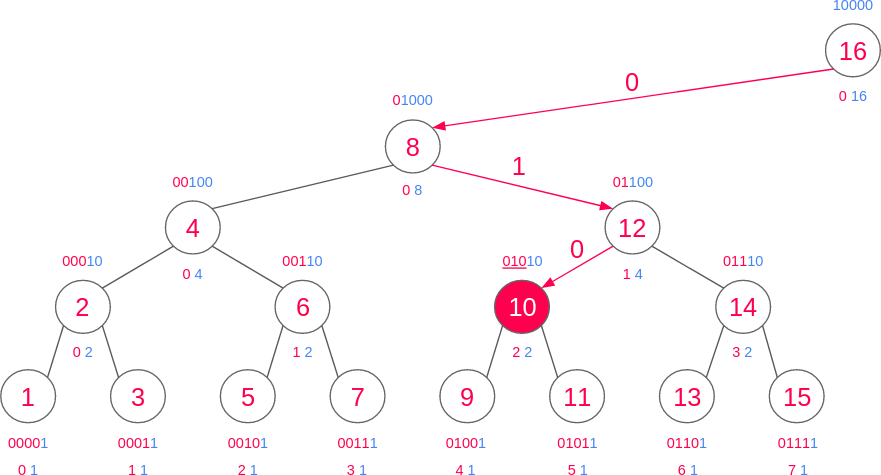


****Figure 9:**** All nodes in the same level have the same suffix.

All nodes in the same level having the same suffix means that all numbers that end with that suffix must appear in that level. *The prefix merely indicates the order of that index in the level.*

For example, the prefix of 2 = (000|10)₂ is 0, which means that 2 is the first element of the level. The prefix of 6= (001|10)₂ is 1, which means that 6 appears at that level after 2. The same goes for 10= (010|10)₂ and so on.

In other words, the prefixes of the same level form the sequence 0, 1, 2, 3…



****Figure 10:****The prefix reveals how to walk on the tree.

We use zeros and ones in binary representation because it is easier to express the idea of two different things, but there is no rule stopping us from using other representations. We could use letters A and B, emojis

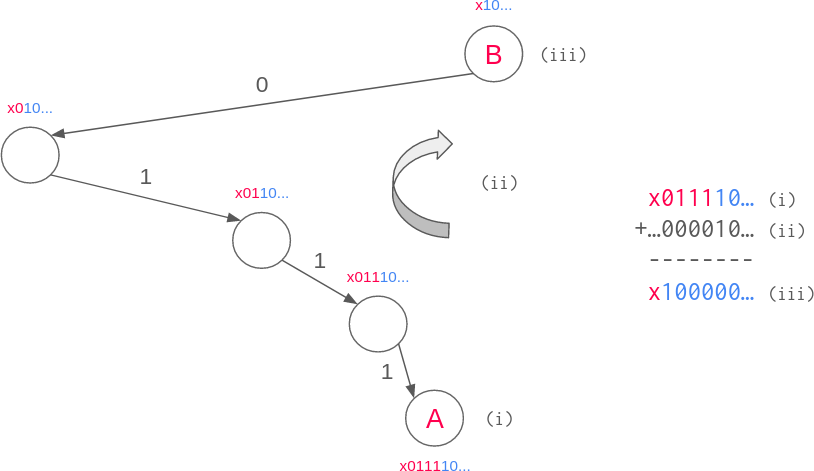
*The Fenwick Tree arranges binary numbers in two dimensions.* Assuming we always start at the root, it makes perfect sense to represent one digit as “go to the left” and the other as “go to the right.” See figure 10.

****The update operation****

*Let A and B be two nodes whose indexes start with the same digits in theirs binary representations and len(prefix(A)) < len(prefix(B)). Then, A is an ancestor of B.*For example, 6=(001|10)₂ is an ancestor of 5=(0010|1)₂ and 7=(0011|1)₂, and 11=(0101|1)₂ is a decedent of 10=(010|10)₂, 12=(01|100)₂, 8=(0|1000)₂. It works just like a [trie](https://en.wikipedia.org/wiki/Trie" \t "https://medium.com/carpanese/_blank). That is an important property to understand the update operation.

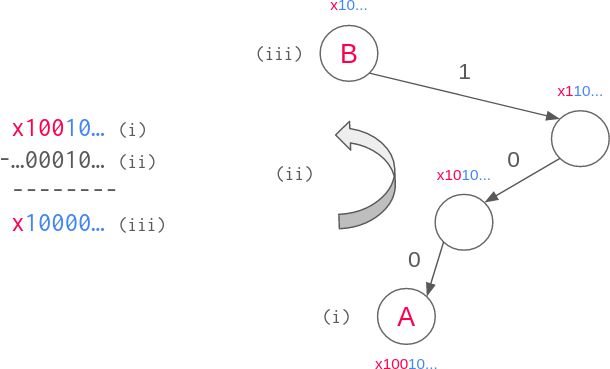
We can say that an ancestor A is above B if it is possible to down from B to A through one or more zeros and then one or more ones. B is immediately above A if it is possible to go down from B to A through exactly one zero and then one or more ones.

*To update a value of the array, we need to move from a given node B to node A immediately above it. Although not intuitive, this is precisely the sum of an index with its least significant bit.*The sum of an index to its least significant bit expresses “go upwards on the tree and stop after arriving at the first 0-labeled edge”. See figure 10a.



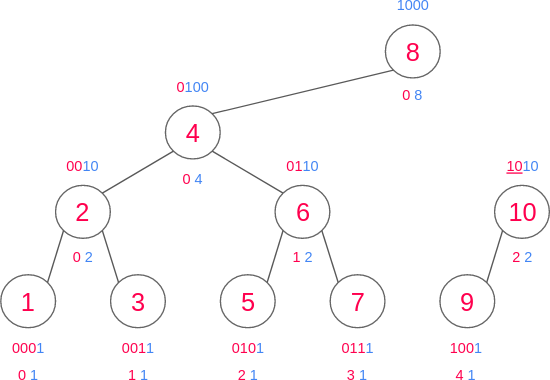
****Figure 10a:****The sum of A with its least significant bit teleports you to node B immediately above A. The “x” means that it could be any sequence of zeros and ones.

That idea of “teleportation” on the tree is another way to understand the query operation. See figure 10b.



****Figure 10b:****The difference of A and its least significant bit teleports you to the node that contains the next segment of the sum of query(A). The “x” means that it could be any sequence of zeros and ones.

Arrays whose size is a power of 2 help us to understand Fenwick Tree easier, but we never needed to use this property. The Fenwick Tree works precisely the way we discussed whatever the size of the array.



****Figure 11a:****An example of a Fenwick Tree with 10 nodes.

## **Implementation**

#include <bits/stdc++.h>

using namespace std;

#define lsb(n) (n & (-n))

struct FenwickTree {

vector<int> v;

vector<int> fenwick;

int e = 0; // neutral element

FenwickTree(vector<int> v) {

this->v = v;

fenwick.assign(v.size()+1, e);

for (int i = 1; i <= v.size(); i++) {

add(i, v[i-1]);

}

}

int op(int a, int b) {

return a + b;

}

int query(int n) {

int res = 0;

while (n > 0) {

res = op(res, fenwick[n]);

n -= lsb(n);

}

return res;

}

int query(int l, int r) {

l++, r++;

if (l == 1) return query(r);

else return query(r) - query(l-1);

}

void add(int i, int diff) {

while (i < fenwick.size()) {

fenwick[i] += diff;

i += lsb(i);

}

}

void update(int i, int val) {

add(i+1, val - v[i]);

v[i] = val;

}

};

int main(void) {

// vector<int> v = {4, 8, 5, 2, 6, 1, 0, 8, 1, 5, 4, 9, 1, 0, 6, 6};

vector<int> v = {-5, 7, 0, 1, 3, 2, -1, 0, 2};

FenwickTree bit = FenwickTree(v);

for (int i = 1; i < bit.fenwick.size(); i++) {

printf("%d\n", bit.fenwick[i]);

}

printf("%d\n", bit.query(1, 4));

bit.update(3, 5);

printf("%d\n", bit.query(1, 4));

}

**Code 4:** Fenwick Tree C++ implementation.

It’s quite easy to analyze the time complexity of this implementation.

The query and update operations are clearly O(lg(*n*)) because we’re iterating through the bits of *n*.

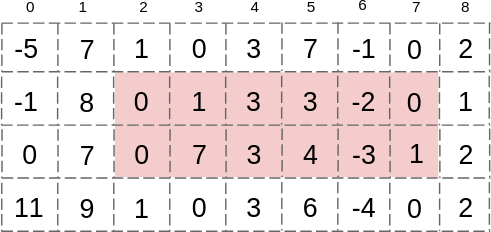
The construction of the Fenwick Tree is O(*n* lg(*n*))call update(i) for *0 ≤ i ≤ n*.

## **Exercises**

****2.**** Prove that the Fenwick Tree is a binary search tree on the indexes.

# **2D Fenwick Tree**

We’ll not explore the variants of the Fenwick Tree in this article, but it’s worth to mention one specific generalization.



****Figure 2:****The range sum query on a 2D array. The sum of the selected area is 17.

What if we wanted to evaluate the sum of a sub-matrix of a given matrix? That’s what the 2D Fenwick Tree tries to solve.