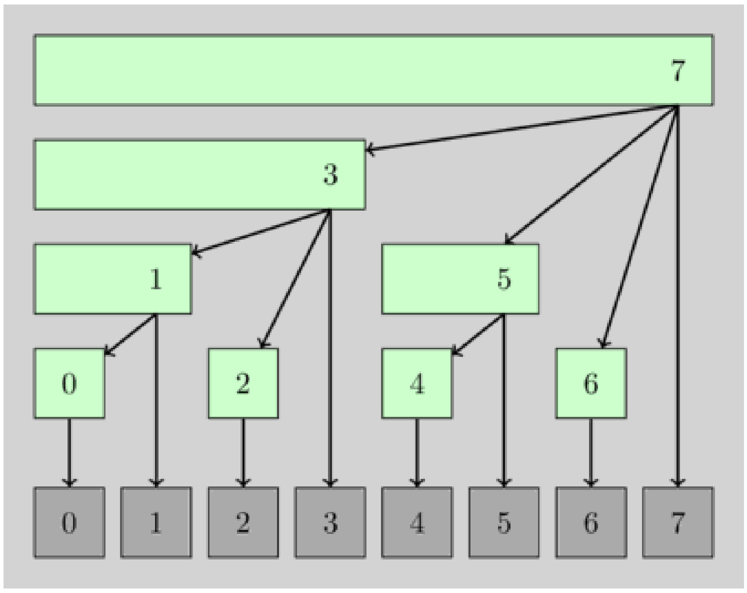

Implementation

Finding sum in one-dimensional array

Here we present an implementation of the Fenwick tree for sum queries and single updates.

The normal Fenwick tree can only answer sum queries of the type $[0, r]$ using `sum(int r)`, however we can also answer other queries of the type $[l, r]$ by computing two sums $[0, r]$ and $[0, l - 1]$ and subtract them. This is handled in the `sum(int l, int r)` method.

Also this implementation supports two constructors. You can create a Fenwick tree initialized with zeros, or you can convert an existing array into the Fenwick form.

```
struct FenwickTree {
    vector<int> bit;  // binary indexed tree
    int n;

    FenwickTree(int n) {
        this->n = n;
        bit.assign(n, 0);
    }

    FenwickTree(vector<int> const &a) : FenwickTree(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }

    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }
    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};
```

Linear construction

The above implementation requires $O(N \log N)$ time. It's possible to improve that to $O(N)$ time.

The idea is, that the number $a[i]$ at index i will contribute to the range stored in $bit[i]$, and to all ranges that the index $i | (i + 1)$ contributes to. So by adding the numbers in order, you only have to push the current sum further to the next range, where it will then get pushed further to the next range, and so on.

```
FenwickTree(vector<int> const &a) : FenwickTree(a.size()){
    for (int i = 0; i < n; i++) {
        bit[i] += a[i];
        int r = i | (i + 1);
        if (r < n) bit[r] += bit[i];
    }
}
```

Finding minimum of $[0, r]$ in one-dimensional array

It is obvious that there is no easy way of finding minimum of range $[l, r]$ using Fenwick tree, as Fenwick tree can only answer queries of type $[0, r]$. Additionally, each time a value is `update`'d, the new value has to be smaller than the current value. Both significant limitations are because the min operation together with the set of integers doesn't form a group, as there are no inverse elements.

```
struct FenwickTreeMin {
    vector<int> bit;
    int n;
    const int INF = (int)1e9;

    FenwickTreeMin(int n) {
        this->n = n;
        bit.assign(n, INF);
    }

    FenwickTreeMin(vector<int> a) : FenwickTreeMin(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            update(i, a[i]);
    }

    int getmin(int r) {
        int ret = INF;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret = min(ret, bit[r]);
        return ret;
    }

    void update(int idx, int val) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] = min(bit[idx], val);
    }
};
```

Note: it is possible to implement a Fenwick tree that can handle arbitrary minimum range queries and arbitrary updates. The paper Efficient Range Minimum Queries using Binary Indexed Trees describes such an approach. However with that approach you need to maintain a second binary indexed tree over the data, with a slightly different structure, since one tree is not enough to store the values of all elements in the array. The implementation is also a lot harder compared to the normal implementation for sums.

Finding sum in two-dimensional array

As claimed before, it is very easy to implement Fenwick Tree for multidimensional array.

```
struct FenwickTree2D {
    vector<vector<int>> bit;
    int n, m;

    // init(...) { ... }

    int sum(int x, int y) {
        int ret = 0;
        for (int i = x; i >= 0; i = (i & (i + 1)) - 1)
            for (int j = y; j >= 0; j = (j & (j + 1)) - 1)
                ret += bit[i][j];
        return ret;
    }

    void add(int x, int y, int delta) {
        for (int i = x; i < n; i = i | (i + 1))
            for (int j = y; j < m; j = j | (j + 1))
                bit[i][j] += delta;
    }
};
```

One-based indexing approach

For this approach we change the requirements and definition for $T[]$ and $g()$ a little bit. We want $T[i]$ to store the sum of $[g(i) + 1; i]$. This changes the implementation a little bit, and allows for a similar nice definition for $g(i)$:

```
def sum(int r):
    res = 0
    while (r > 0):
        res += t[r]
        r = g(r)
    return res

def increase(int i, int delta):
    for all j with g(j) < i <= j:
        t[j] += delta
```

The computation of $g(i)$ is defined as: toggling of the last set 1 bit in the binary representation of i.

$$g(7) = g(111_2) = 110_2 = 6$$

$$g(6) = g(110_2) = 100_2 = 4$$

$$g(4) = g(100_2) = 000_2 = 0$$

The last set bit can be extracted using $i \ \& \ (-i)$, so the operation can be expressed as:

$$g(i) = i - (i \ \& \ (-i)).$$

And it's not hard to see, that you need to change all values $T[j]$ in the sequence $i,\ h(i),\ h(h(i)),\ \dots$ when you want to update $A[j]$, where $h(i)$ is defined as:

$$h(i) = i + (i \ \& \ (-i)).$$

As you can see, the main benefit of this approach is that the binary operations complement each other very nicely.

The following implementation can be used like the other implementations, however it uses one-based indexing internally.

```
struct FenwickTreeOneBasedIndexing {
    vector<int> bit;  // binary indexed tree
    int n;

    FenwickTreeOneBasedIndexing(int n) {
        this->n = n + 1;
        bit.assign(n + 1, 0);
    }

    FenwickTreeOneBasedIndexing(vector<int> a)
        : FenwickTreeOneBasedIndexing(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    int sum(int idx) {
        int ret = 0;
        for (++idx; idx > 0; idx -= idx & -idx)
            ret += bit[idx];
        return ret;
    }

    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }

    void add(int idx, int delta) {
        for (++idx; idx < n; idx += idx & -idx)
            bit[idx] += delta;
    }
};
```

Range operations

A Fenwick tree can support the following range operations:

1. Point Update and Range Query
2. Range Update and Point Query
3. Range Update and Range Query

1. Point Update and Range Query

This is just the ordinary Fenwick tree as explained above.

2. Range Update and Point Query

Using simple tricks we can also do the reverse operations: increasing ranges and querying for single values.

Let the Fenwick tree be initialized with zeros. Suppose that we want to increment the interval $[l, r]$ by x. We make two point update operations on Fenwick tree which are `add(l, x)` and `add(r+1, -x)`.

If we want to get the value of $A[i]$, we just need to take the prefix sum using the ordinary range sum method. To see why this is true, we can just focus on the previous increment operation again. If $i < l$, then the two update operations have no effect on the query and we get the sum 0. If $i \in [l, r]$, then we get the answer x because of the first update operation. And if $i > r$, then the second update operation will cancel the effect of first one.

The following implementation uses one-based indexing.

```
void add(int idx, int val) {
    for (++idx; idx < n; idx += idx & -idx)
        bit[idx] += val;
}

void range_add(int l, int r, int val) {
    add(l, val);
    add(r + 1, -val);
}

int point_query(int idx) {
    int ret = 0;
    for (++idx; idx > 0; idx -= idx & -idx)
        ret += bit[idx];
    return ret;
}
```

Note: of course it is also possible to increase a single point $A[i]$ with `range_add(i, i, val)`.

3. Range Update and Range Query

To support both range updates and range queries we will use two BITs namely $B_1[]$ and $B_2[]$, initialized with zeros.

Suppose that we want to increment the interval $[l, r]$ by the value x. Similarly as in the previous method, we perform two point updates on B_1: `add(B1, l, x)` and `add(B1, r+1, -x)`. And we also update B_2. The details will be explained later.

```
def range_add(l, r, x):
    add(B1, l, x)
    add(B1, r+1, -x)
    add(B2, l, x*(l-1))
    add(B2, r+1, -x*r))
```

After the range update (l, r, x) the range sum query should return the following values:

$$sum[0, i] = \begin{cases} 0 & i < l \\ \\ x \cdot (i - (l-1)) & l \le i \le r \\ \\ x \cdot (r - l + 1) & i > r \end{cases}$$

We can write the range sum as difference of two terms, where we use B_1 for first term and B_2 for second term. The difference of the queries will give us prefix sum over $[0, i]$.

$$\begin{aligned} sum[0, i] &= sum(B_1, i) \cdot i - sum(B_2, i) \\ &= \begin{cases} 0 \cdot i - 0 & i < l \\ \\ x \cdot i - x \cdot (l-1) & l \le i \le r \\ \\ 0 \cdot i - (x \cdot (l-1) - x \cdot r) & i > r \end{cases} \end{aligned}$$

The last expression is exactly equal to the required terms. Thus we can use B_2 for shaving off extra terms when we multiply $B_1[i] \times i$.

We can find arbitrary range sums by computing the prefix sums for $l-1$ and r and taking the difference of them again.

```
def add(b, idx, x):
    while idx <= N:
        b[idx] += x
        idx += idx & -idx

def range_add(l,r,x):
    add(B1, l, x)
    add(B1, r+1, -x)
    add(B2, l, x*(l-1))
    add(B2, r+1, -x*r)

def sum(b, idx):
    total = 0
    while idx > 0:
        total += b[idx]
```