Exercise 2.3.4: Draw a Segment Tree of this array `A = {10, 2, 47, 3, 7, 9, 1, 98, 21, 37}` and answer RMQ(1, 7) and RMQ(3, 8)!

Exercise 2.3.5: Using the same Segment Tree as in exercise 1 above, answer this Range Sum Query(i, j) (RSQ), i.e. a sum from `A[i] + A[i + 1] + ... + A[j]`. What is `RSQ(1, 7)` and `RSQ(3, 8)`? Is this a good approach to solve this problem? (See Section 3.4.2).

Exercise 2.3.6: The Segment Tree code shown above lacks the `update` operation. Add the $O(\log n)$ `update` function to update the value of a certain index in array `A` (and its Segment Tree)!

2.3.4 Fenwick Tree

Fenwick Tree (also known as Binary Indexed Tree or BIT) was invented by *Peter Fenwick* in 1994 [9]. Fenwick Tree is a useful data structure for implementing *cumulative frequency tables*. For example, if we have test scores of 10 students `s = {2,4,5,5,6,6,6,7,7,8}`, then the frequency of each score $[0 \ldots 8]$ and their cumulative frequency can be seen in Table 2.1 (with comments).

Index/ Score	Frequency	Cumulative Frequency	Short Comment
0	0	0	Let's abbreviate Cumulative Frequency as 'CF'
1	0	0	(CF for score 1) = 0 + 0 = 0.
2	1	1	(CF for score 2) = 0 + 0 + 1 = 1.
3	0	1	(CF for score 3) = 0 + 0 + 1 + 0 = 1.
4	1	2	(CF for score 4) = 0 + 0 + 1 + 0 + 1 = 2.
5	2	4	...
6	3	7	...
7	2	9	...
8	1	10	(CF for score 8) = (CF for score 7) + 1 = 9 + 1 = 10.

Table 2.1: Example of a Cumulative Frequency Table

This cumulative frequency table is akin to the Range Sum Query (RSQ) mentioned in Section 2.3.3 where now we perform incremental `RSQ(0, i)` $\forall i \in [0 \ldots 8]$, i.e. using the example above, then `RSQ(0, 0) = 0`, ..., `RSQ(0, 2) = 1`, ..., `RSQ(0, 6) = 7`, ..., `RSQ(0, 8) = 10`.

Rather than implementing Segment Tree to implement such cumulative frequency table, it is *far easier* to code Fenwick Tree instead (please compare the two sample source codes provided in this section versus in Section 2.3.3). Perhaps this is one of the reason why Fenwick Tree is currently included in the IOI syllabus [10].

Fenwick Trees are typically implemented as arrays (our library uses `vector`). Fenwick Tree is a tree that is indexed by the bits of its *integer* keys. These integer keys must have a fixed range (in programming contest setting, this can go up to 1M – large enough for many practical problems). Each index is responsible for certain range as shown in Figure 2.8. Let the name of Fenwick Tree array as `ft`. Then, `ft[3]` is responsible for range `[0..3]`, `ft[5]` is responsible for range `[4..5]`, `ft[6]` is responsible for range `[6..6]`, etc[12].

With such arrangements, if we want to know the cumulative frequency between two indices `[0..b]`, i.e. `ft_rsq(ft, 0, b)`, we simply add `ft[b]`, `ft[b']`, `ft[b'']`, ... until the array index is < 0. This sequence of indices are obtained via this bit manipulation: `b' = (b & (b + 1)) - 1`. In Figure 2.8, the `ft_rsq(t, 0, 6) = ft[6] + ft[5] + ft[3] = 3 + 3 + 1 = 7`. Notice that index 3, 5, and 6 are responsible for range `[0..3]`, `[4..5]`, and `[6..6]`, respectively. These 3 indices 6, 5, 3 are related in their binary form: `b` $= 6_{10} =$ `(110)`$_2$ can be transformed to `b'` $= 5_{10} =$ `(101)`$_2$ and subsequently to `b''` $= 3_{10} =$ `(011)`$_2$ via bit manipulation above. As an integer b only has $O(\log b)$ bits, then this `ft_rsq(ft, 0, b)` is just $O(\log n)$ when `n = b`.

[12]In this book, we will not go into details why such arrangements work. Interested readers are advised to read [9].

Now, to get the cumulative frequency between two indices `[a..b]` where `a != 0` is equally simple, just do `ft_rsq(ft, 0, b) - ft_rsq(ft, 0, a - 1)`. In Figure 2.8, the `ft_rsq(ft, 3, 6) = ft_rsq(ft, 0, 6) - ft_rsq(ft, 0, 2) = 7 - 1 = 6`. Again, this is just an $O(\log n)$ operation.

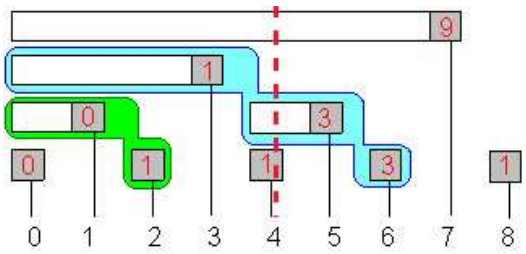

Figure 2.8: Example of a Fenwick Tree

Then if we want to update the value of a certain index, for example if we want to adjust the value for index `k` by certain value `v` (v can be positive or negative), i.e. `ft_adjust(ft, k, v)`, then we have to update `ft[k]`, `ft[k']`, `ft[k'']`, ... until array index exceeds the size of array. This sequence of indices are obtained via this bit manipulation: `k' = k | (k + 1)`. In Figure 2.8, `ft_adjust(ft, 4, 1)` will affect `ft` at indices `k` = 4_{10} = $(100)_2$, `k'` = 5_{10} = $(101)_2$, and `k''` = 7_{10} = $(111)_2$. These series of indices can be obtained with bit manipulation above. Notice that if you project a line upwards from index 4 in Figure 2.8, you will see that line intersects the range handled by index 4, index 5, and index 7. Starting from any integer `k`, this operation `ft_adjust(ft, k, v)` just needs at most $O(\log n)$ steps to make `k` $\geq$ `n` where `n` is the size of the Fenwick Tree.

Fenwick Tree needs $O(n)$ memory and $O(\log n)$ access/update operations for a set of n keys. This makes Fenwick Tree an ideal data structure for solving *dynamic* RSQ problem (recall that the *static* RSQ problem can be solved with $O(n)$ DP + $O(1)$ per query).

```
#include <iostream>
#include <vector> // recall that we use this shorcut: typedef vector<int> vi;
using namespace std;

vi ft_create(int n) { return vi(n, 0); } // initially n elements, all zeroes.

int ft_rsq(const vi &t, int a, int b) { // returns RSQ(a, b)
  if (a == 0) { int sum = 0; for (; b >= 0; b = (b & (b + 1)) - 1) sum += t[b]; return sum; }
  else return ft_rsq(t, 0, b) - ft_rsq(t, 0, a - 1);
}

// adjusts value of the k-th element by v (v can be positive/increase or negative/decrease).
void ft_adjust(vi &t, int k, int v) { for (; k < (int)t.size(); k |= k + 1) t[k] += v; }

int main() {            // idx   0 1 2 3 4 5 6 7 8
  vi ft = ft_create(9); // ft = {0,0,0,0,0,0,0,0,0}
  ft_adjust(ft, 2, 1);  // ft = {0,0,1,1,0,0,0,1,0}, index 2,3,7 are affected (all +1)
  ft_adjust(ft, 4, 1);  // ft = {0,0,1,1,1,1,0,2,0}, index 4,5,7 are affected (all +1)
  ft_adjust(ft, 5, 2);  // ft = {0,0,1,1,1,3,0,4,0}, index 5,7 are affected (all +2)
  ft_adjust(ft, 6, 3);  // ft = {0,0,1,1,1,3,3,7,0}, index 6,7 are affected (all +3)
  ft_adjust(ft, 7, 2);  // ft = {0,0,1,1,1,3,3,9,0}, only index 7 is affected (+2)
  ft_adjust(ft, 8, 1);  // ft = {0,0,1,1,1,3,3,9,1}, only index 8 is affected (+1)
  printf("%d\n", ft_rsq(ft, 0, 0)); // returns 0 , ft[0] (idx 0)
  printf("%d\n", ft_rsq(ft, 0, 2)); // returns 1 , ft[2] (idx 2) + ft[1] (idx 0-1)
  printf("%d\n", ft_rsq(ft, 0, 6)); // returns 7 , ft[6] (idx 6) + ft[5] (idx 4-5) + ft[3] (idx 0-3)
  printf("%d\n", ft_rsq(ft, 0, 8)); // returns 10, ft[8] (idx 8) + ft[7] (idx 0-7)
  printf("%d\n", ft_rsq(ft, 3, 6)); // returns 6 , rsq(ft, 0, 6) - rsq(ft, 0, 2) = 7 - 1 = 6.
  return 0;
}
```

Example codes: `ch2_09_fenwicktree_ds.cpp`; `ch2_09_fenwicktree_ds.java`

Exercise 2.3.7: Fenwick Tree has an additional operation: find the index with a given cumulative frequency. For example we want to know what is the min score in Table 2.1 so that there are at least 7 students covered (in this case index 6). Hint: the cumulative frequency is sorted, thus we can use *binary search* (see Section 3.2).

Exercise 2.3.8: Extend the 1-D Fenwick Tree to 2-D!

Programming exercises that use data structures with our own libraries:

- Graph Data Structures Problems
  1. *UVa 10720 - Graph Construction* (uses Erdos-Gallai's theorem)
  2. *UVa 10895 - Matrix Transpose* (transpose adjacency list)
  3. *UVa 10928 - My Dear Neighbours* (counting out degrees)
  4. *UVa 11414 - Dreams* (similar to UVa 10720; Erdos-Gallai's theorem)
  5. *UVa 11550 - Demanding Dilemma* (graph representation, incidence matrix)
  6. Much more graph problems are discussed in Chapter 4
- Union-Find Disjoint Sets
  1. UVa 00459 - Graph Connectivity (also solvable with 'flood fill' in Section 4.2)
  2. *UVa 00599 - The Forrest for the Trees* (|trees/acorns| related to disjoint set size)
  3. UVa 00793 - Network Connections (trivial; application of disjoint sets)
  4. UVa 10158 - War (advanced disjoint sets)
  5. *UVa 10227 - Forests* (merge two disjoint sets if they are consisten)
  6. *UVa 10507 - Waking up brain* (using disjoint sets simplifies this problem)
  7. UVa 10583 - Ubiquitous Religions (count the remaining disjoint sets after unions)
  8. UVa 10608 - Friends (find the set with largest element)
  9. *UVa 10685 - Nature* (find the set with largest element)
  10. UVa 11474 - Dying Tree
  11. UVa 11503 - Virtual Friends (maintain set attribute (size) in representative item)
  12. *UVa 11690 - Money Matters* (check if sum of money from each member of a set is 0)
- Segment and Fenwick Tree
  1. UVa 11235 - Frequent Values (range max query)
  2. UVa 11297 - Census (2D Segment Tree/Quad Tree)
  3. UVa 11402 - Ahoy, Pirates! (requires updates to the Segment Tree)
  4. UVa 11525 - Permutation (can use Fenwick tree + binary search)
  5. *UVa 11610 - Reverse Prime* (reverse primes $< 10^6$, append zero(s), use FT + bsearch)
  6. LA 2191 - Potentiometers (Dhaka06) (range sum query)
  7. LA 3294 - The Ultimate Bamboo Eater (Dhaka05) (toposort, DP, 2D Segment Tree)
  8. LA 4108 - Skyline (Singapore07) (uses Segment Tree)

Profile of Algorithm Inventor

Rudolf Bayer (May 7, 1939) has been Professor (emeritus) of Informatics at the Technical University of Munich since 1972.

He is famous for inventing two data sorting structures: the B-tree with Edward M. McCreight, and later the UB-tree with Volker Markl. He also invented the red-black trees which is typically used in C++ STL `map` and `set`.