MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
A Constituent Institution of Manipal University

Department of Computer Science and Engineering

Subject: Advanced Data Structures & Algorithms Lab

Subject Code: CSE – 5141

Project Topic: Fenwick Tree Algorithm (Binary Indexed Tree)

Presented by:

Pasupuleti Rohith Sai Datta (230913003)

Chinmaya D Kamath (230913006)

Under the guidance of:

Prof. Gururaj

Introduction:

In the realm of computer science and algorithmic problem-solving, the efficient manipulation of data structures is paramount. One of the perennial challenges faced in this pursuit is the computation of range sums within an array, a problem ubiquitous in various applications ranging from database systems to statistical analysis. Traditional methods often involve linear-time complexity, making them less than ideal for large datasets. Enter the Fenwick Tree, a data structure designed to address this concern and elevate the efficiency of range sum queries and updates.

This report delves into the Fenwick Tree, exploring its intricacies, applications, and advantages over conventional methods. As we navigate through the conceptual foundations and operational principles of the Fenwick Tree, we aim to provide a comprehensive understanding of why and how it stands as a powerful tool in the arsenal of data structure solutions.

Let, f be some group operation (binary associative function over a set with identity element and inverse elements) and A be an array of integers of length N.

Fenwick tree is a data structure which:

- calculates the value of function f in the given range $[l, r]$ (i.e. $f(A_l, A_{l+1}, \dots, A_r)$) in $O(\log N)$ time; updates the value of an element of A in $O(\log N)$ time; requires $O(N)$
- memory, or in other words, exactly the same memory required for A; is easy to use and code, especially, in the case of multidimensional arrays.
- The most common application of Fenwick tree is *calculating the sum of a range* (i.e. using
- addition over the set of integers Z: $f(A_1, A_2, \dots, A_k) = A_1 + A_2 + \dots + A_k$).

Fenwick tree is also called Binary Indexed Tree, or just BIT abbreviated.

Description:

For the sake of simplicity, we will assume that function f is just a *sum function*.

Given an array of integers $A[0 \dots N-1]$. A Fenwick tree is just an array $T[0 \dots N-1]$, where each of its elements is equal to the sum of elements of A in some range $[g(i), i]$:

$$T_i = \sum_{j=g(i)}^{i} A_j,$$

where g is some function that satisfies $0 \leq g(i) \leq i$. We will define the function in the next few paragraphs.

The data structure is called a tree, because there is a nice representation of the data structure as tree, although we don't need to model an actual tree with nodes and edges. We will only need to maintain the array T to handle all queries.

Note: The Fenwick tree presented here uses zero-based indexing. Many people will actually use a version of the Fenwick tree that uses one-based indexing. Therefore you will also nd an alternative implementation using one-based indexing in the implementation section. Both versions are equivalent in terms of time and memory complexity.

Now we can write some pseudo-code for the two operations mentioned above - get the sum of elements of A in the range $[0, r]$ and update (increase) some element A_i:

```
def sum(int r):
    res = 0
    while (r >= 0):
        res += t[r]
        r = g(r) - 1
    return res

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

The function Sum works as follows:
1. First, it adds the sum of the range [g(r), r] (i.e., T[r]) to the result.
2. Then, it "jumps" to the range [g(g(r) − 1), g(r) − 1], and adds this range's sum to the result.
3. And so on, until it "jumps" from [0, g(g(… g(r) − 1 − 1) − 1)] to [g(−1), −1]; that is where the sum function stops jumping.

The function 'increase' works with the same analogy but "jumps" in the direction of increasing indices:
1. Sums of the ranges [g(j), j] that satisfy the condition g(j) ≤ i ≤ j are increased by delta, i.e., t[j] += delta. Therefore, we update all elements in T that correspond to ranges in which Ai lies.

It is obvious that the complexity of both sum and increase depends on the function g. There are many ways to choose the function g, as long as $0 \le g(i) \le i$ for all i. For instance, the function g(i) = i works, resulting in T = A, and therefore summation queries are slow. We can also take the function g(i) = 0. This will correspond to prefix sum arrays, meaning that finding the sum of the range [0, i] will only take constant time, but updates are slow. The clever part of the Fenwick algorithm is that it uses a special definition of the function g that can handle both operations in O(logN) time.

Definition of g(i):
The computation of g(i) is defined using the following simple operation: we replace all trailing 1 bits in the binary representation of i with 0 bits. In other words, if the least significant digit of i in binary is 0, then g(i) = i. Otherwise, the least significant digit is a 1, and we take this 1 and all other trailing 1s and flip them. For instance, we get:

$g(11) = g(1011_2) = 1000_2 = 8$
$g(12) = g(1100_2) = 1100_2 = 12$
$g(13) = g(1101_2) = 1100_2 = 12$
$g(14) = g(1110_2) = 1110_2 = 14$
$g(15) = g(1111_2) = 0000_2 = 0$

There exists a simple implementation using bitwise operations for the non-trivial operation described above:

[g(i) = i & (i + 1)]

where & is the bitwise AND operator. It is not hard to convince yourself that this solution does the same thing as the operation described above. Now, we just need to find a way to iterate over all j's such that g(j) ≤ i ≤ j. It is easy to see that we can find all such j's by starting with i and flipping the last unset bit. We will call this operation h(j). For example, for i = 10, we have:

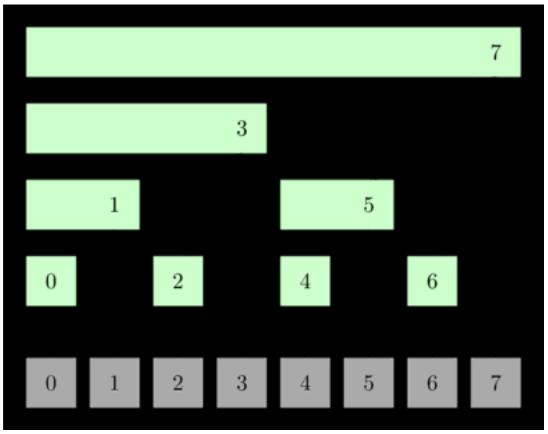
$[10 = 0001010_2]$
$[h(10) = 11 = 0001011_2]$
$[h(11) = 15 = 0001111_2]$

[h(15) = 31 = 0011111$_2$]
[h(31) = 63 = 0111111$_2$]

Unsurprisingly, there also exists a simple way to perform h using bitwise operations:

[h(j) = j parallel (j + 1),]

where (parallel) is the bitwise OR operator. The following image shows a possible interpretation of the Fenwick tree as a tree. The nodes of the tree show the ranges they cover.



Problem Description:

Let's start with a mutable range sum query problem. Given an array of numbers A = \{A_1, A_2, ..., A_n\} with starting index of 1, we have two operations: update and sum. For an update operation, update(i , value), we change the value of the element at the index i to be value. For a sum operation, sum(left , right), we calculate the sum of numbers between positions left and right inclusive. Our goal is to implement these two operations efficiently.

Brute Force Solution:

For the update operation, we can directly update the array element at the specified location. For the sum operation, we can iterate the array from index left to right and sum each element:

Algorithm 1: Brute Force Approach to Solve Mutable Range Sum Query Problem

```
Data: A number array A with size n
Function update(i, value):
    A[i] = value;
Function sum(left, right):
    result = 0;
    for i = left to right do
        result += A[i] ;
    end
    return result;
```

In this solution, the update operation takes O(1) time and sum operation takes O(n) time. The space requirement is O(1).

Prefix Sum Solution:

In order to have a faster sum operation, we can use the prefix sum approach. For the input array of numbers A = \{A_1, A_2, ..., A_n\}, the prefix sum at index i is the sum of the prefix numbers ending at i:

Algorithm 2: Construct a Prefix Sum Array

```
Data: A number array A with size n
Result: The prefix sum array of A
Function prefixSum(A, n):
    prefix[0] = 0;
    for k = 1 to n do
        prefix[k] = prefix[k − 1] + A[k];
    end
    return prefix ;
```

For the sum operation, we can directly answer the query with two prefix sum values. However, we need to update the prefix sum array for the update operation:

Algorithm 3: Prefix Sum Approach to Solve Mutable Range Sum Query Problem

```
Data: A number array A with size n
Function update(i, value):
    delta = value − A[i];
    for k = i to n do
        prefix[k] += delta;
    end
    A[i] = value;
Function sum(left, right):
    return prefix[right] − prefix[left − 1];
```

In this solution, the update operation takes O(n) time and sum operation takes O(1) time. The space requirement is O(n) as we need an array to store prefix sums.

Fenwick Tree Solution:

For the brute force and prefix sum solutions, either update or sum operation takes O(n) time. If there are multiple update and sum operations, these two solutions will take a lot of time. For example, if we have O(n) numbers of update and sum operations, the overall time complexity will be O(n^2) for both solutions.

We can use a Fenwick tree to solve this problem more efficiently, as it only takes O(logn) time for both update and sum operations. Therefore, if we have O(n) numbers of update and sum operations, the overall time complexity will be O(nlogn).

Range of Responsibility:

When we make an update operation, we have an input index parameter, i, to indicate the update element location. Similarly, when we make a prefixSum operation, we have an input index parameter, i, to indicate the sum of the prefix numbers ending at i. In the Fenwick tree solution, we do both operations based on the binary form of the input index i. That's the reason why we also call it a binary index tree.

In a Fenwick tree, each index has a range of responsibilities. We calculate the range of responsibility value based on the position of the rightmost set bit (RSB) in the binary representation of the index. For example, the binary representation of 11 is (1011)_2. The range of responsibility of 11 is then 1 as its RSB gives value (1)_2 = 1. Another example of a range of responsibilities is for the number 12 =(1100)_2. The range of responsibility of 12 is 4 as its RSB gives value (100)_2 = 4.

Range of Responsibility Calculation:

In computer science, we use the two's complement method to produce a negative number from a positive number. Firstly, we reverse the ones and zeros of the binary representation of the positive number. Then we add one. Therefore, we can calculate the range of responsibility value of an index I with bit operation i & (-i).

For example, the binary value of 12 in an 8-bit form is (0000, 1100)_
After we reverse each bit, we have (1111\,0011)
When we add 1, every bit 1 starting at the right will overflow into the next higher bit until we reach a 0 bit, which is the RSB of the original positive number. Therefore, we can have -12 = (1111, 0100).

When we do bitwise AND operation on the two numbers, we can get the range of responsibility value: 12 ,& ,(-12) = (0000, 1100)_2 ,& ,(1111, 0100)_2 = (0000 ,0100)_2 = 4.
Let's define the range of responsibility calculation, RSB, for an index i:

Algorithm 4: Range of Responsibility Calculation

Data: An index number i
Result: Range of responsibility of i
Function RSB(i):
  return $i \,\&\, (-i)$;

Fenwick Tree Structure:

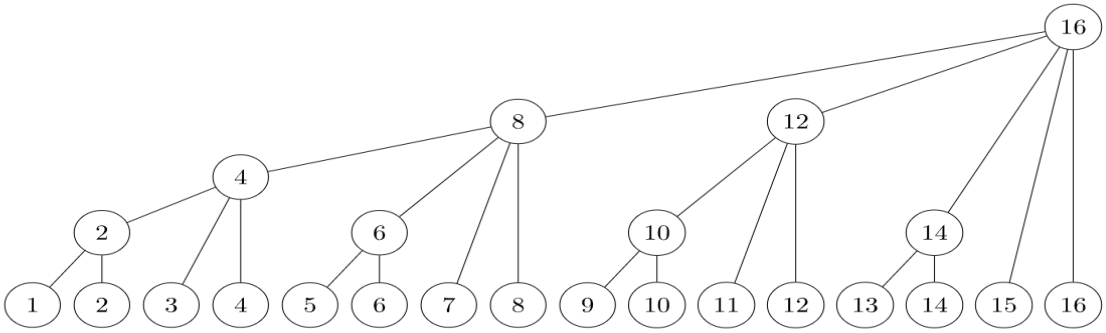
The basic idea of the Fenwick tree solution is to precompute the responsibility range sum for each index. Then, we can calculate the prefix sum of an index i based on the precomputed Fenwick values. Also, when we update the element at an index i, we can only update those precomputed Fenwick values that cover the index i.

$$Fenwick[i] = \sum_{k=i-RSB(i)+1}^{i} A_k$$

Let Fenwick[i] denote the responsibility range sum for each index i. For the array A = \{A_1, A_2, ..., A_n}, we can have have an array Fenwick where
The following table shows the Fenwick values for numbers from 1 to 16:

Number	Binary Representation	RSB	Fenwick
1	$(0000\,0001)_2$	$(1)_2 = 1$	$sum(A_1)$
2	$(0000\,0010)_2$	$(10)_2 = 2$	$sum(A_1, A_2)$
3	$(0000\,0011)_2$	$(1)_2 = 1$	$sum(A_3)$
4	$(0000\,0100)_2$	$(100)_2 = 4$	$sum(A_1, A_2, A_3, A_4)$
5	$(0000\,0101)_2$	$(1)_2 = 1$	$sum(A_5)$
6	$(0000\,0110)_2$	$(10)_2 = 2$	$sum(A_5, A_6)$
7	$(0000\,0111)_2$	$(1)_2 = 1$	$sum(A_7)$
8	$(0000\,1000)_2$	$(1000)_2 = 8$	$sum(A_1, A_2, ..., A_8)$
9	$(0000\,1001)_2$	$(1)_2 = 1$	$sum(A_9)$
10	$(0000\,1010)_2$	$(10)_2 = 2$	$sum(A_9, A_{10})$
11	$(0000\,1011)_2$	$(1)_2 = 1$	$sum(A_{11})$
12	$(0000\,1100)_2$	$(100)_2 = 4$	$sum(A_9, A_{10}, A_{11}, A_{12})$
13	$(0000\,1101)_2$	$(1)_2 = 1$	$sum(A_{13})$
14	$(0000\,1110)_2$	$(10)_2 = 2$	$sum(A_{13}, A_{14})$
15	$(0000\,1111)_2$	$(1)_2 = 1$	$sum(A_{15})$
16	$(0001\,0000)_2$	$(1\,0000)_2 = 16$	$sum(A_1, A_2, ..., A_{16})$

Based on the Fenwick value definition, we can see that one Fenwick value can cover several Fenwick values. For example, Fenwick[8] = sum(A_1,A_2,...,A_8) covers Fenwick[4]=sum(A_1, A_2, A_3, A_4), Fenwick[6] = sum(A_5, A_6) and Fenwick[7] = sum(A_7). We can use a tree structure to represent the coverage relationships:



In this Fenwick tree structure, leaf nodes are the index numbers. Each internal node represents a Fenwick value that can be constructed from its children's Fenwick values and its own index array number.

For example:

[Fenwick[16] =Fenwick[8]+Fenwick[12]+Fenwick[14]+Fenwick[15]+A_{16}]

Fenwick Tree Construction:

To construct a Fenwick tree, we can use a dynamic programming approach by propagating the smaller Fenwick values to the big Fenwick values:

Algorithm 5: Construct a Fenwick Tree

```
Data: A number array A with size n
Result: The Fenwick array of A
Function constructFenwick(A, n):
    fenwick = A;
    for i = 1 to n do
        parent = i + RSB (i);
        if parent ≤ n then
            fenwick[parent]+= fenwick[i];
        end
    end
    return fenwick ;
```

In this algorithm, we first initialize the Fenwick array with the input array A. Then, we use a loop to propagate the smaller Fenwick values to their parent Fenwick values. The time complexity is O(n).

We only need to construct the Fenwick array once at the beginning. In the future, we only need to update at most O(logn) Fenwick values in an update operation.

Prefix Sum Calculation:

When we compute the prefix sum of index i, we can start from index i and go downwards until we reach 0:

Algorithm 6: Calculate Prefix Sum From a Fenwick Tree

```
Data: An index i
Result: The prefix sum of A ending at index i
Function prefixSum(i):
    result = 0;
    while i > 0 do
        result+=fenwick[i];
        i -= RSB (i);
    end
    return result ;
```

In this algorithm, we start from the input index i and add its Fenwick value to the result. Then, we remove the RSB of i and add the new Fenwick value into the result. We keep doing this process until we reach index 0.

Suppose we want to find the prefix sum for index 11 = (1011)_2. Firstly, we add Fenwick[11] into our result. Then, we subtract the RSB value of 11, which is 1, and reach a new index 10=(1010)_2. Therefore, we add Fenwick[10] into our result. Again, we subtract RSB(10) = 2 from 10 and get a new index 8=(1000)_2. Similarly, we add Fenwick[8] into our result. Finally, we subtract RSB(8)=8 from the current index 8 and stop the loop as we reach the index 0.

In the end, we can have prefixSum(11) = Fenwick[11] + Fenwick[10] + Fenwick[8].
The time complexity of this algorithm is O(logn) as we remove one set bit of the original index i in each iteration until we reach 0.

Fenwick Tree Update:

The update operation is the opposite of the prefixSum operation. We start from the input index i and go upwards along the tree path to propagate the update operation.
Suppose we want to add a delta value at the location 3. We can first add delta to fenwick[3]. Then, we get its parent based on the RSB(3) and reach the index 4. Therefore we also add delta to fenwick[4]. Similarly, we continue to update the parent Fenwick values like Fenwick[8] and Fenwick[16] until we reach the array size boundary n.

Algorithm 7: Update a Fenwick Tree

```
Data: An index i and delta to add
Function add(i, delta):
    while i ≤ n do
        fenwick[i] += delta;
        i += RSB (i);
    end
```

Similar to the prefixSum, the time complexity of add is O(logn).

Fenwick Tree Solution:

We can combine the above functions to solve the mutable range sum query problem:

Algorithm 8: Fenwick Tree Approach to Solve Mutable Range Sum Query Problem

```
Data: A number array A with size n
Function update(i, value):
    delta = value − A[i];
    add (i, delta);
    A[i] = value;
Function sum(left, right):
    return prefixSum (right) - prefixSum (left − 1);
```

In the update function, we first calculate the delta of the updated value, Then, we call the add function to update the Fenwick tree. The time complexity is O(logn) as we only call the add function once.
In the sum function, we call prefixSum function twice to get the sum between left and right. The overall time complexity is still O(logn).
The space complexity of the Fenwick solution is O(n) as we need to store the Fenwick values for all the indexes.

Example of Fenwick tree:

Binary Indexed Tree also called Fenwick Tree provides a way to represent an array of numbers in an array, allowing prefix sums to be calculated efficiently. For example, an array [2, 3, -1, 0, 6] is given, then the prefix sum of first 3 elements [2, 3, -1] is 2 + 3 + -1 = 4. Calculating prefix sum efficiently is useful in various scenarios. Let's start with a simple problem.
Given an array, and two types of operations are to be performed on it.
Change the value stored at an index i. (This is called a point update operation)
Find the sum of a prefix of length k. (This is called a range sum query)

This is the last set bit,
and we need to isolate this.

Let's take an example, a number x = 1110(in binary),

Binary digit	1	1	1	0
Index	3	2	1	0

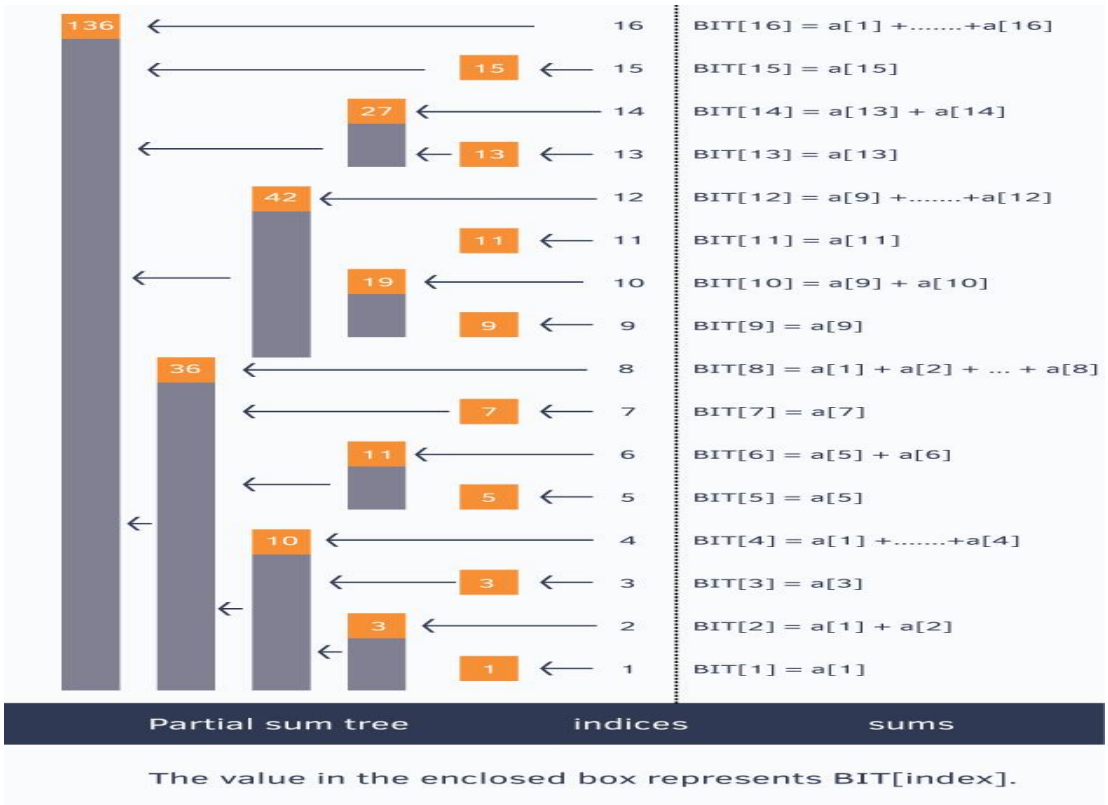
Say x=a1b(in binary) is the number whose last set bit we want to isolate.
Here is some binary sequence of any length of 1's and 0's and is some sequence of any length but of 0's only. Remember we said we want the LAST set bit, so for that tiny intermediate 1 bit sitting between a and b to be the last set bit, b should be a sequence of 0's only of length zero or more.
-x = 2's complement of x = (a1b)' + 1 = a'0b' + 1 = a'0(0….0)' + 1 = a'0(1…1) + 1 = a'1(0…0) = a'1b

```
      a1b      <-- This is x
&     a¯1b     <-- This is -x
————————
= (0...0)1(0...0)   <-- This is the last set bit isolated.
```

Example: x = 10(in decimal) = 1010(in binary)
The last set bit is given by x&(-x) = (10)1(0) & (01)1(0) = 0010 = 2
(in decimal) But why do we need to isolate this weird last set bit in any number? Well, we will be seeing that as you proceed further.Now let's dive into Binary Indexed tree.

Basic Idea of Binary Indexed Tree:
We know the fact that each integer can be represented as the sum of powers of two. Similarly, for a given array of size , we can maintain an array such that, at any index we can store the sum of some numbers of the given array. This can also be called a partial sum tree.
Let's use an example to understand how stores partial sums.

```
int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
```



The value in the enclosed box represents BIT[index].

```
                {           a[x],                   if x is odd
BIT[x] =                     a[1] + ... + a[x],      if x is power of 2
                }
```

Query Operation in Binary Indexed Tree (BIT):

Suppose we have a BIT and want to perform a query operation, such as query(14):

Initialize Variables:
Start with sum = 0 as the initial sum value.

Iterative Process:
We want to find the prefix sum up to index 14.
In each iteration, isolate the last set bit from the current index (14 in this case) and add the corresponding BIT value to the sum.

Iteration 1:
Isolate the last set bit in 14 (which is 2) and subtract it from 14.
Last set bit in 14 is 2, so subtracting gives 12.
Add the BIT value at index 14 to the sum.

Iteration 2:
Isolate the last set bit in 12 (which is 4) and subtract it from 12.
Last set bit in 12 is 4, so subtracting gives 8.
Add the BIT value at index 12 to the sum.

Iteration 3:
Isolate the last set bit in 8 (which is 8) and subtract it from 8.
Last set bit in 8 is 8, so subtracting gives 0.
Add the BIT value at index 8 to the sum.
The loop stops because the last set bit in 0 is not found.

Result:
The sum variable now holds the prefix sum up to index 14.
Time Complexity:The loop iterates at most the number of bits in the given index (14 in this case), which is at most log(14) ~ 4 times.
Therefore, the query operation takes O(log N) time, where N is the size of the array represented by the BIT.This process efficiently calculates the prefix sum using the properties of Binary Indexed Trees.

When to Use Binary Indexed Tree (BIT):

Operation Requirements:
Associative: The operation should be associative, meaning the result is the same regardless of how elements are grouped. This applies to both BIT and segment tree.
Has Inverse: The operation should have an inverse. Examples include addition and subtraction, multiplication and division.
No Inverse for Certain Operations:
Some operations, like finding the greatest common divisor (gcd), don't have inverses. In such cases, BIT is not suitable for calculating range gcd.
Space and Time Complexity:
Space Complexity: BIT requires extra space for an array of the same size as the input.
Time Complexity: Both update and query operations take logarithmic time.

Applications:
Arithmetic Coding Algorithm: BIT is used to implement the arithmetic coding algorithm. It's chosen for its ability to efficiently support specific operations.
Counting Inversions: BIT is helpful for counting inversions in an array efficiently.

Pseudocode:

```
function update(fenwickTree, n, index, value):
   index = index + 1
   while index <= n:
      fenwickTree[index] += value
      index += index & -index  # Move to the next node in the tree

function query(fenwickTree, index):
   index = index + 1
   sum = 0
   while index > 0:
      sum += fenwickTree[index]
      index -= index & -index  # Move to the parent node in the tree
   return sum

function findKth(fenwickTree, n, k):
   index = 0
   mask = 1 << 30  # Assuming integers are 32 bits
   while mask != 0 and index < n:
      tIndex = index + mask
      if tIndex <= n and k > fenwickTree[tIndex]:
         index = tIndex
         k -= fenwickTree[tIndex]
      mask >>= 1  # Move to the next bit
   return index

function rangeUpdate(fenwickTree, n, start, end, value):
   update(fenwickTree, n, start, value)
   update(fenwickTree, n, end + 1, -value)

# Example usage:
n = 10
fenwickTree = array of size n + 1 initialized with zeros

# Update values
update(fenwickTree, n, 2, 3)
update(fenwickTree, n, 5, 2)
update(fenwickTree, n, 1, 5)

# Query operations
sumInRange = query(fenwickTree, 5) - query(fenwickTree, 2 - 1)
kthElement = findKth(fenwickTree, n, 3 + 1)
rangeUpdate(fenwickTree, n, 2, 8, 2)
```

Explanation:

The update function increments the Fenwick Tree values starting from a given index with a specified value.
The query function calculates the sum of values in the prefix up to a given index.
The findKth function finds the index of the kth element in the Fenwick Tree.
The rangeUpdate function updates a range of values in the Fenwick Tree efficiently.

In the example usage:
sumInRange calculates the sum in the range [2, 5].
kthElement finds the index of the 3rd element in the Fenwick Tree.
rangeUpdate updates the values in the range [2, 8] efficiently.

Time complexity analysis of Fenwick tree:

Update Operation:

Time Complexity: O(log n)
When updating a value in the original array, the Fenwick Tree modifies nodes along a tree path. The time complexity is logarithmic, based on the binary representation of the index.

Query Operation:

Time Complexity: O(log n)
Computing the cumulative sum up to a specific index involves visiting nodes determined by the binary representation of the index. This results in a logarithmic time complexity.

Range Update Operation:

Time Complexity: O(log n)
Achieving a range update is done through two point updates, maintaining a logarithmic time complexity for the overall operation.

Find Kth Element Operation:

Time Complexity: O(log n)
The operation to find the index of the kth element also exhibits a logarithmic time complexity. It iteratively traverses the tree, adjusting the search based on the value of k.

Amortised analysis for Fenwick tree:

1. **Aggregate Analysis:**

In Aggregate Analysis, we divide the total cost of a sequence of operations by the number of operations.

Update Operation:

Each update in Fenwick Tree takes worst-case O(log n) time.
For n update operations, total cost is O(n log n).
Amortized cost per update: O(log n).
Query Operation:
Similar to update, query takes worst-case O(log n) time.
Total cost for n queries: O(n log n).
Amortized cost per query: O(log n).

2. **Potential Method:**

This method defines a potential function representing "extra" work during an operation.
Potential Function:
Let Φ(t) be the potential function at time t.
Φ(t) = ∑ Ci - Actual Cost(i)

Update Operation:
Actual cost is O(log n).
Potential increases by O(log n).
Amortized cost: O(log n).
Query Operation:
Amortized cost of a query: O(log n).

3. **Accounting Method:**

Here, we assign "credits" to operations to offset costlier ones.

Update Operation:
Charge O(log n) for each update.
Credits pay for potential increase in the next update.
Amortized cost: O(log n).
Query Operation:
Amortized cost of a query: O(log n).

Analysis Conclusion:
All methods agree: Amortized cost for both update and query in Fenwick Tree is O(log n).
Fenwick Trees efficiently handle dynamic cumulative frequency calculations with logarithmic amortized complexity per operation.
Note: Analysis assumes Fenwick Tree starts with zeros and input indices are valid. The amortized analysis provides insight into average cost over a sequence, enhancing our understanding of the data structure's efficiency.

Output:

```
Enter the size of the array: 5
Enter the number of update operations: 5
Enter index and value for update operation 1: 1 1
Enter index and value for update operation 2: 2 2
Enter index and value for update operation 3: 3 3
Enter index and value for update operation 4: 4 4
Enter index and value for update operation 5: 5 5
Enter the number of queries: 3
Enter query type (1 for range query, 2 for find kth, 3 for sum up to index) and corresponding parameters for query 1: 1
Enter start and end indices for range query: 2 4
Sum in the range [2, 4]: 9
Enter query type (1 for range query, 2 for find kth, 3 for sum up to index) and corresponding parameters for query 2: 2
Enter the value of k for find kth query: 3
3th element in the Fenwick tree: 3
Enter query type (1 for range query, 2 for find kth, 3 for sum up to index) and corresponding parameters for query 3: 3
Enter the index for sum up to index query: 4
Sum up to index 4: 10
```

Conclusion:

In summary, Fenwick Tree, or Binary Indexed Tree (BIT), is a compact data structure designed for efficient point updates and prefix sum queries in an array. Leveraging binary indexing, it achieves logarithmic time complexity for these operations. Its applications range from cumulative frequency tables to scenarios requiring fast-range queries, making it a valuable tool in algorithmic problem-solving.