Experiment No. 3

Title: Implementation of Fenwick Tree

Batch:A2 **Roll No:16010421059** **Experiment No.:3**

Aim:

Resources needed: Text Editor, C/C++ IDE

Theory:

Binary Indexed Tree or 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 is [2, 3, -1, 0, 6] the length 3 prefix [2, 3, -1] with sum 2 + 3 + -1 = 4). Calculating prefix sums efficiently is useful in various scenarios. Let's start with a simple problem.

We are given an array a[], and we want to be able to perform two types of operations on it.

1. Change the value stored at an index i. (This is called a **point update** operation)
2. Find the sum of a prefix of length k. (This is called a **range sum** query)

Simple Solution is :

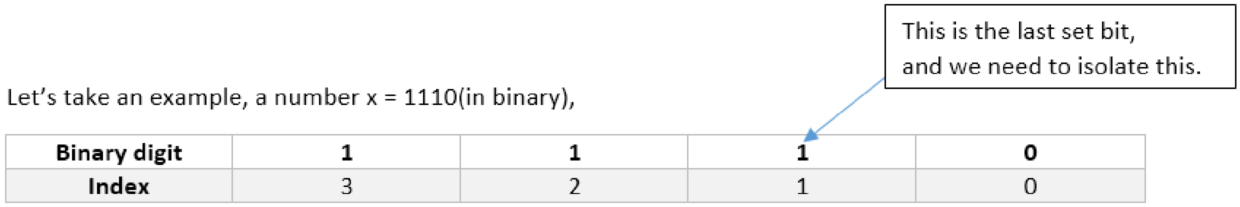
```
int a[] = {2, 1, 4, 6, -1, 5, -32, 0, 1};
void update(int i, int v)   //assigns value v to a[i]
{
    a[i] = v;
}
int prefixsum(int k)     //calculate the sum of all a[i] such that 0 <= i < k
{
   int sum = 0;
   for(int i = 0; i < k; i++)
        sum += a[i];
   return sum;
}
```

But the time required to calculate a prefix sum is proportional to the length of the array, so this will usually time out when a large number of such intermingled operations are performed.

One efficient solution is to use a segment tree that can perform both operations in O(logN) time.

Using binary Indexed tree also, we can perform both the tasks in O(logN) time. But then why learn another data structure when segment tree can do the work for us. It’s because binary indexed trees require less space and are very easy to implement during programming contests

Understanding Bit manipulation :

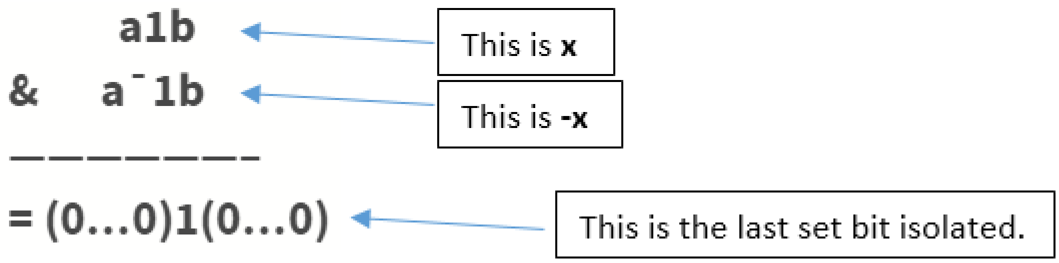


This is the last set bit, and we need to isolate this.

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

Binary digit	**1**	**1**	**1**	**0**
Index	3	2	1	0

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**



$$\begin{array}{ll} & a1b \\ \& & a^{-}1b \\ \hline & = (0...0)1(0...0) \end{array}$$

This is **x**

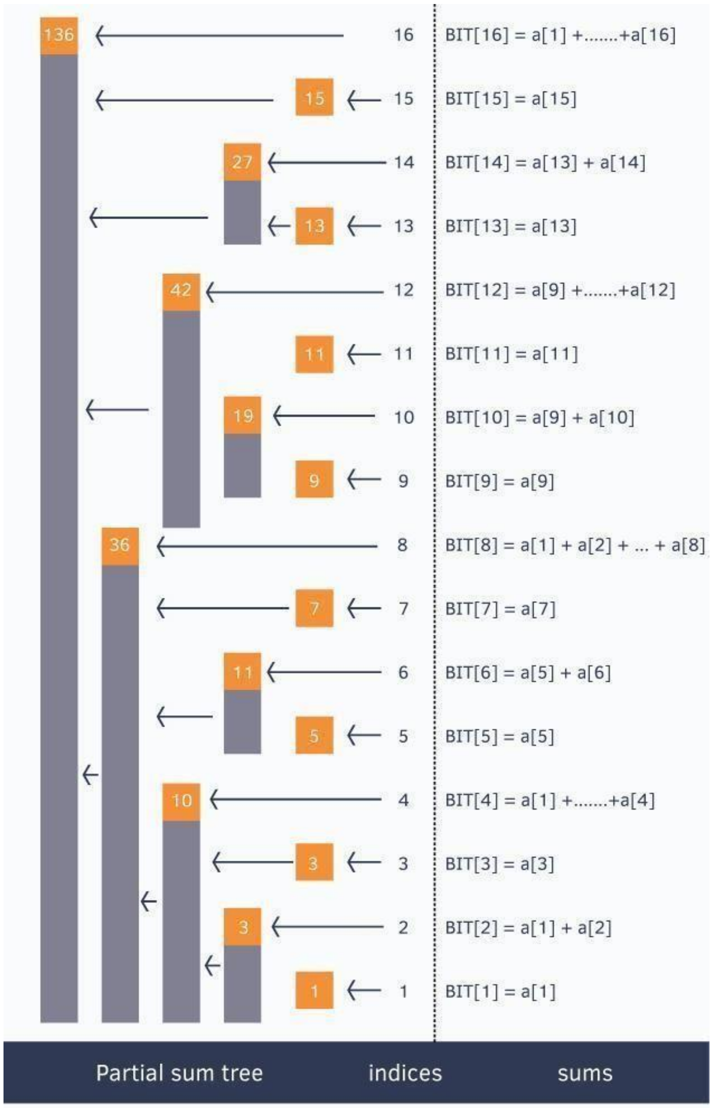
This is **-x**

This is the last set bit isolated.

Basic Idea of Binary Indexed Tree:

We know the fact that each integer can be represented as sum of powers of two. Similarly, for a given array of size N, we can maintain an array BIT[] such that, at any index we can store sum of some numbers of the given array. This can also be called a partial sum tree.

Let Us Consider **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].

The above picture shows the binary indexed tree, each enclosed box of which denotes the value BIT[index] and each BIT[index] stores a partial sum of some numbers.

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

To generalize this every index i in the BIT[] array stores the cumulative sum from the index i to i - (1<<r) + 1 (both inclusive), where r represents the last set bit in the index i

Sum of first 12 numbers in array

a[] = BIT[12] + BIT[8] = (a[12] + … + a[9]) + (a[8] + … + a[1])

sum of first 6 elements = BIT[6] + BIT[4] = (a[6] + a[5]) + (a[4] + … + a[1])

Sum of first 8 elements = BIT[8] = a[8] + … + a[1]

we call update() operation for each element of a given array to construct the Binary Indexed Tree. The update() operation is discussed below.

```
void update(int x, int delta)       //add "delta" at index "x"
{
for(; x <= n; x += x&-x)
 BIT[x] += delta;
}
```

Suppose we call update(13, 2).

Here we see from the above figure that indices 13, 14, 16 cover index 13 and thus we need to add 2 to them also.

Initially x is 13, we update BIT[13]

BIT[13] += 2;

Now isolate the last set bit of x = 13(1101) and add that to x , i.e. x += x&(-x)

Last bit is of x = 13(1101) is 1 which we add to x, then x = 13+1 = 14, we update BIT[14]

BIT[14] += 2;

Now 14 is 1110, isolate last bit and add to 14, x becomes 14+2 = 16(10000), we update BIT[16]

BIT[16] += 2;

How to **query** such structure for prefix sums?

```
int query(int x)      //returns the sum of first x elements in given array a[]
{
     int sum = 0;
     for(; x > 0; x -= x&-x)
         sum += BIT[x];
     return sum;
}
```

The above function query() returns the sum of first x elements in given array. Let’s see how it works.

Suppose we call **query(14)**, initially **sum = 0**

x is 14(1110) we add BIT[14] to our sum variable, thus **sum = BIT[14]** = (a[14] + a[13])

now we isolate the last set bit from **x = 14(1110)** and subtract it from x

last set bit in **14(1110)** is 2(10), thus **x = 14 – 2 = 12**

we add BIT[12] to our sum variable, thus[

sum = BIT[14] + BIT[12] = (a[14] + a[13]) + (a[12] + … + a[9])

again we isolate last set bit from x = 12(1100) and subtract it from x

last set bit in 12(1100) is 4(100), **thus x = 12 – 4 = 8**

we add BIT[8] to our sum variable, thus

sum = BIT[14] + BIT[12] + BIT[8] = (a[14] + a[13]) + (a[12] + … + a[9]) + (a[8] + … + a[1])

once again we isolate last set bit from x = 8(1000) and subtract it from x

last set bit in 8(1000) is 8(1000), **thus x = 8 – 8 = 0**

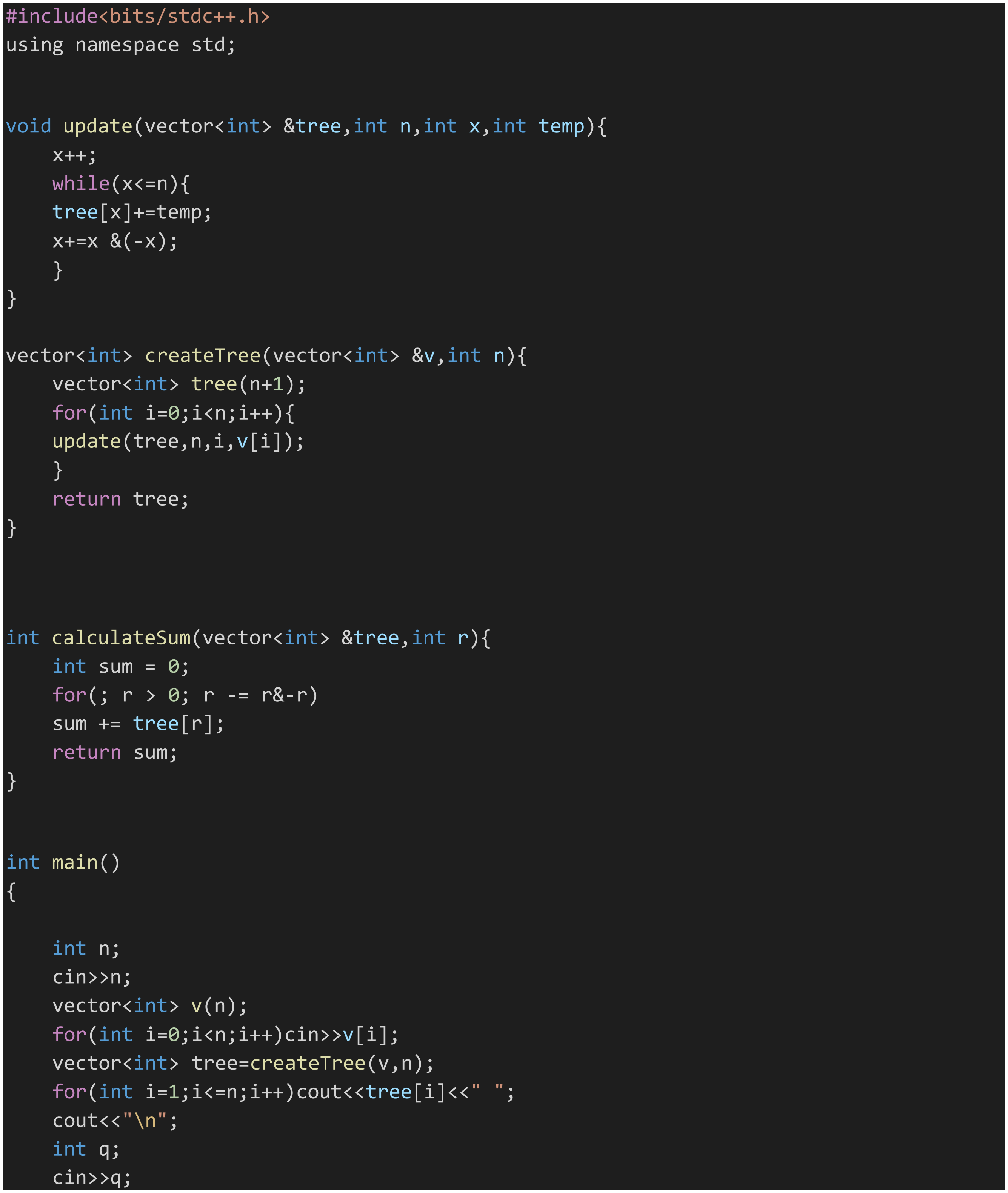
since x = 0, the for loop breaks and we return the prefix sum.

Space Complexity: **O(N)** for declaring another array of size N
Time Complexity: **O(logN)** for each operation(update and query as well)

Activity:

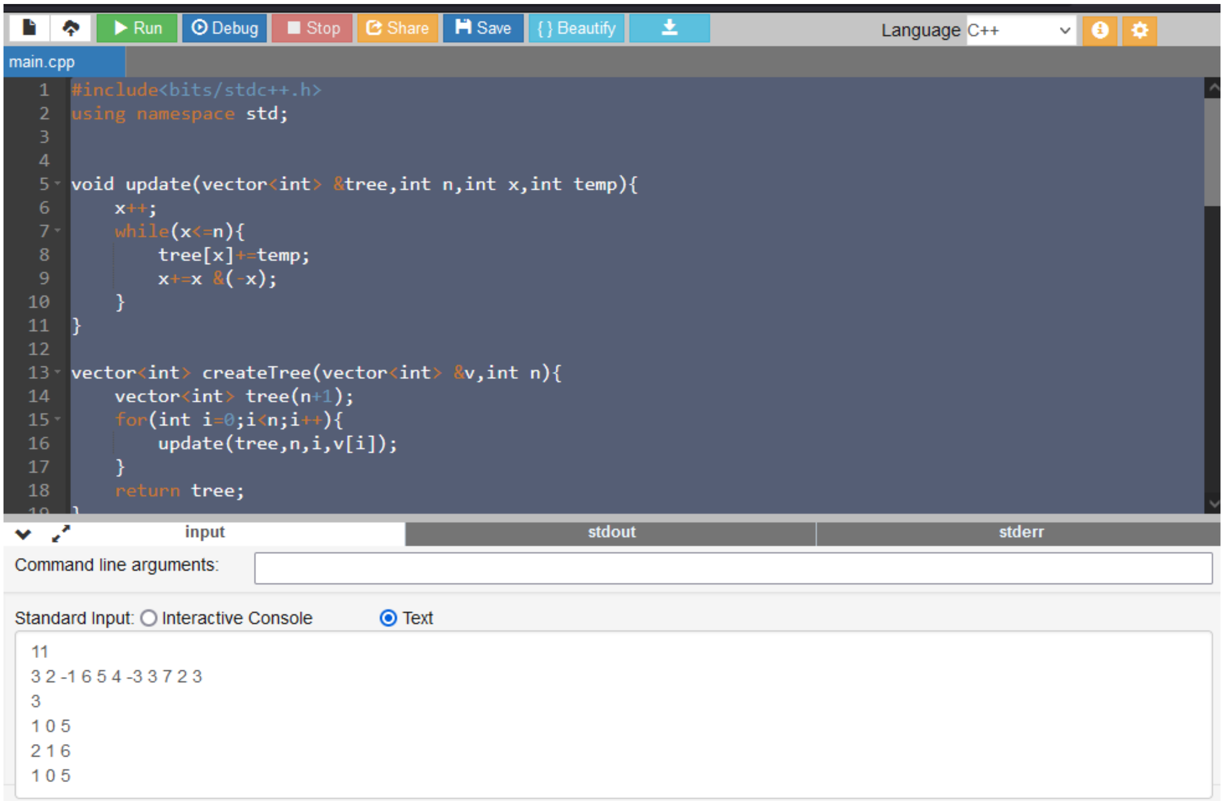
Write a program to solve range-based query over an array for performing sum and update operation using Fenwick tree.

Solution:



```
#include<bits/stdc++.h>
using namespace std;


void update(vector<int> &tree,int n,int x,int temp){
    x++;
    while(x<=n){
    tree[x]+=temp;
    x+=x &(-x);
    }
}

vector<int> createTree(vector<int> &v,int n){
    vector<int> tree(n+1);
    for(int i=0;i<n;i++){
    update(tree,n,i,v[i]);
    }
    return tree;
}



int calculateSum(vector<int> &tree,int r){
    int sum = 0;
    for(; r > 0; r -= r&-r)
    sum += tree[r];
    return sum;
}


int main()
{

    int n;
    cin>>n;
    vector<int> v(n);
    for(int i=0;i<n;i++)cin>>v[i];
    vector<int> tree=createTree(v,n);
    for(int i=1;i<=n;i++)cout<<tree[i]<<" ";
    cout<<"\n";
    int q;
    cin>>q;
    while(q--){
        int t;
        cin>>t;
        if(t==1){
            int l,r;
            cin>>l>>r;
            cout<<calculateSum(tree,r)-calculateSum(tree,l);
            cout<<"\n";
    }
    if(t==2){
        int  i,d;
        cin>>i>>d;
        update(tree,n,i,d);
    }

    }
}
```

Output:

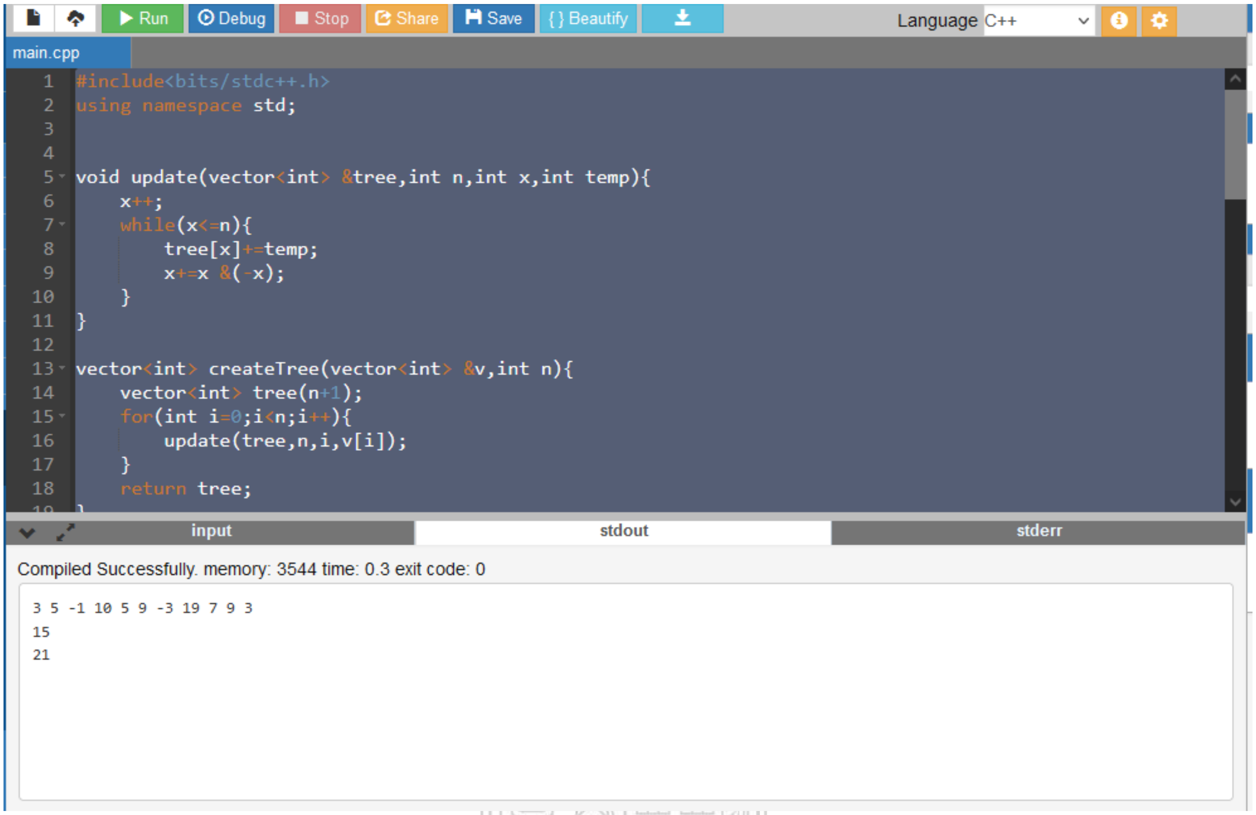


```
#include<bits/stdc++.h>
using namespace std;


void update(vector<int> &tree,int n,int x,int temp){
    x++;
    while(x<=n){
        tree[x]+=temp;
        x+=x &(-x);
    }
}

vector<int> createTree(vector<int> &v,int n){
    vector<int> tree(n+1);
    for(int i=0;i<n;i++){
        update(tree,n,i,v[i]);
    }
    return tree;
```

Command line arguments:

Standard Input: ○ Interactive Console ● Text

```
11
3 2 -1 6 5 4 -3 3 7 2 3
3
1 0 5
2 1 6
1 0 5
```

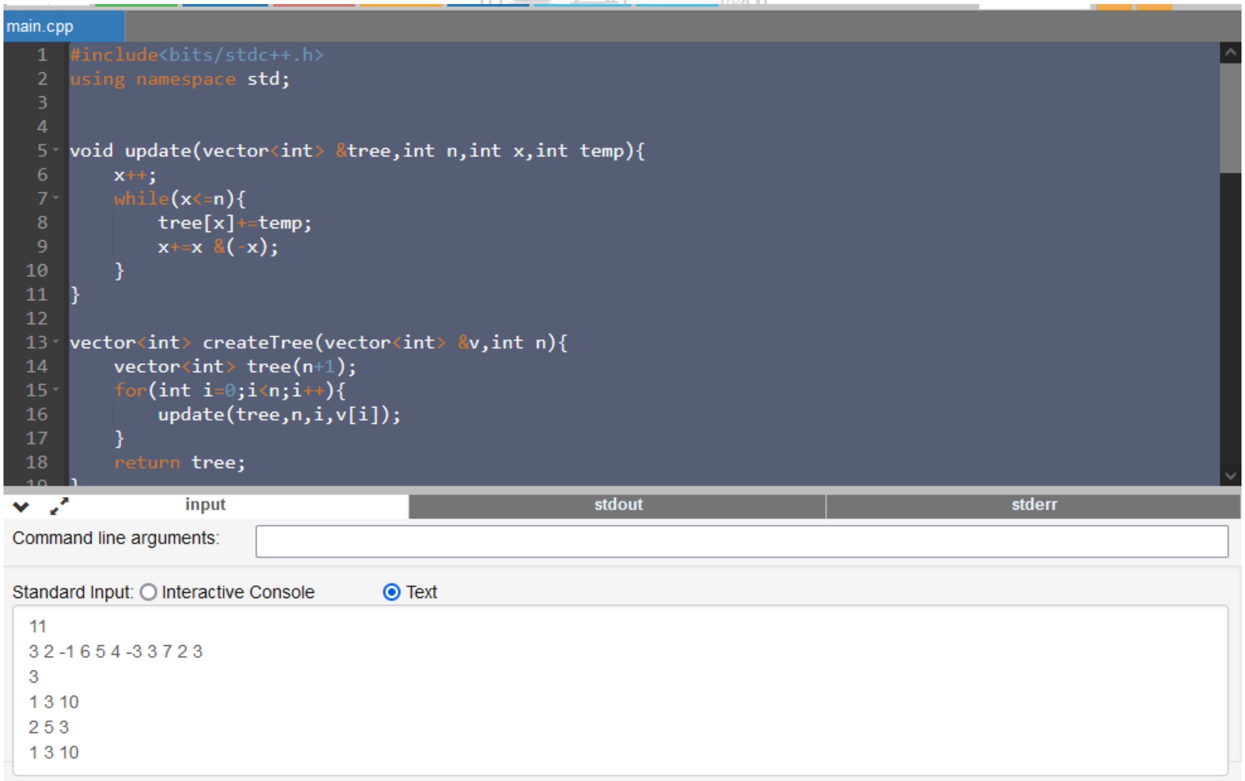


Language C++

```
#include<bits/stdc++.h>
using namespace std;


void update(vector<int> &tree,int n,int x,int temp){
    x++;
    while(x<=n){
        tree[x]+=temp;
        x+=x &(-x);
    }
}

vector<int> createTree(vector<int> &v,int n){
    vector<int> tree(n+1);
    for(int i=0;i<n;i++){
        update(tree,n,i,v[i]);
    }
    return tree;
}
```

Compiled Successfully. memory: 3544 time: 0.3 exit code: 0

```
3 5 -1 10 5 9 -3 19 7 9 3
15
21
```

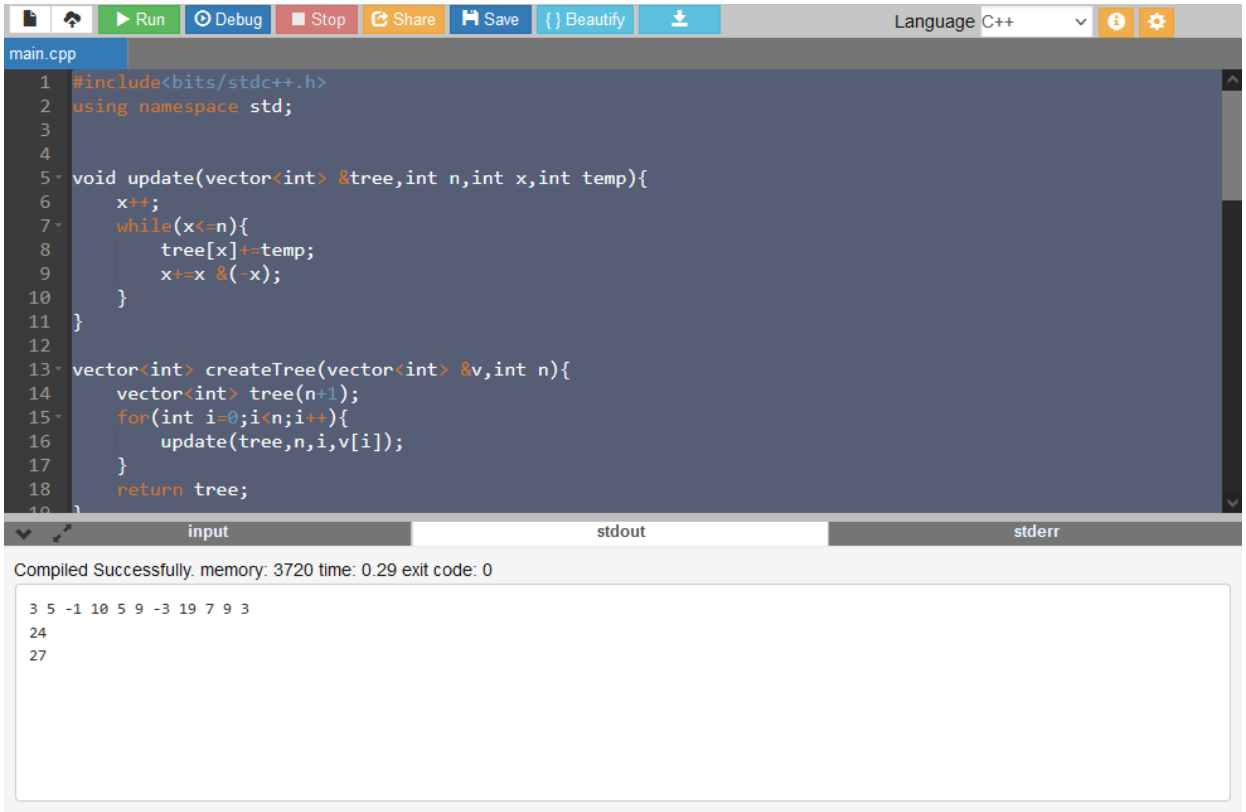


```
#include<bits/stdc++.h>
using namespace std;


void update(vector<int> &tree,int n,int x,int temp){
    x++;
    while(x<=n){
        tree[x]+=temp;
        x+=x &(-x);
    }
}

vector<int> createTree(vector<int> &v,int n){
    vector<int> tree(n+1);
    for(int i=0;i<n;i++){
        update(tree,n,i,v[i]);
    }
    return tree;
}
```

Command line arguments:

Standard Input: ○ Interactive Console ● Text

```
11
3 2 -1 6 5 4 -3 3 7 2 3
3
1 3 10
2 5 3
1 3 10
```

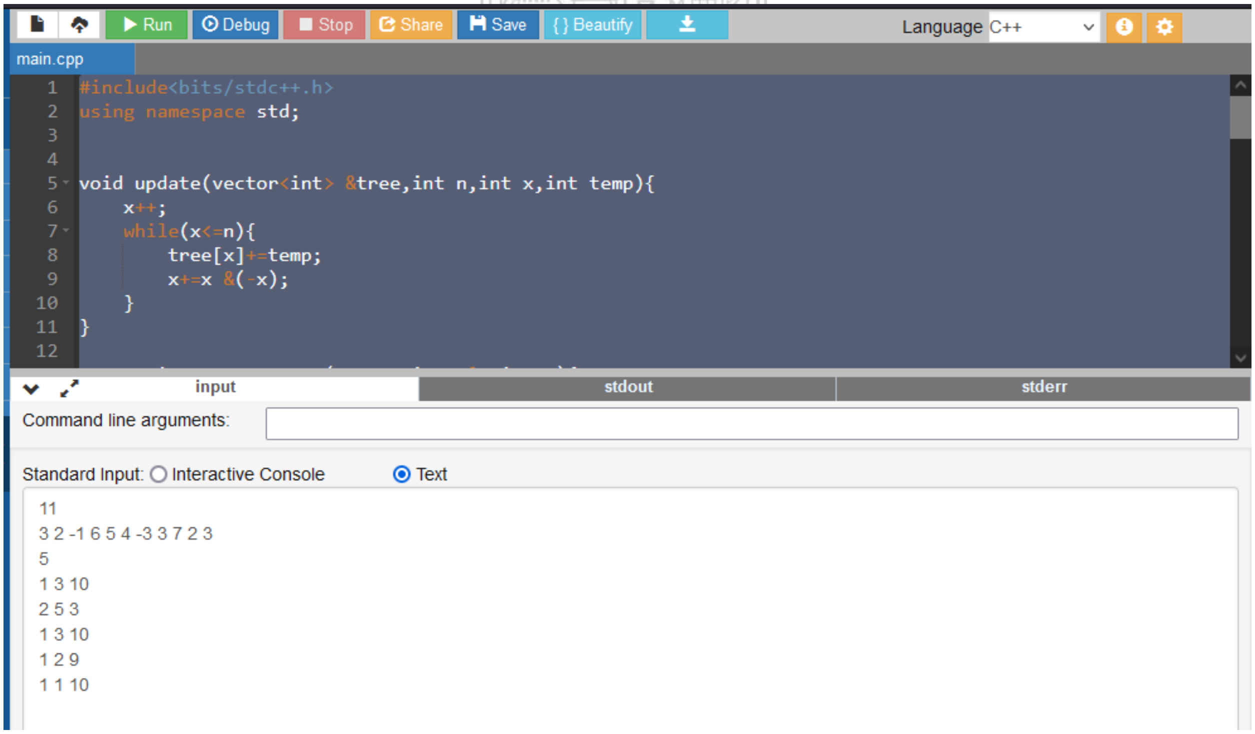


Language C++

```cpp
#include<bits/stdc++.h>
using namespace std;


void update(vector<int> &tree,int n,int x,int temp){
    x++;
    while(x<=n){
        tree[x]+=temp;
        x+=x &(-x);
    }
}

vector<int> createTree(vector<int> &v,int n){
    vector<int> tree(n+1);
    for(int i=0;i<n;i++){
        update(tree,n,i,v[i]);
    }
    return tree;
```

Compiled Successfully. memory: 3720 time: 0.29 exit code: 0

```
3 5 -1 10 5 9 -3 19 7 9 3
24
27
```



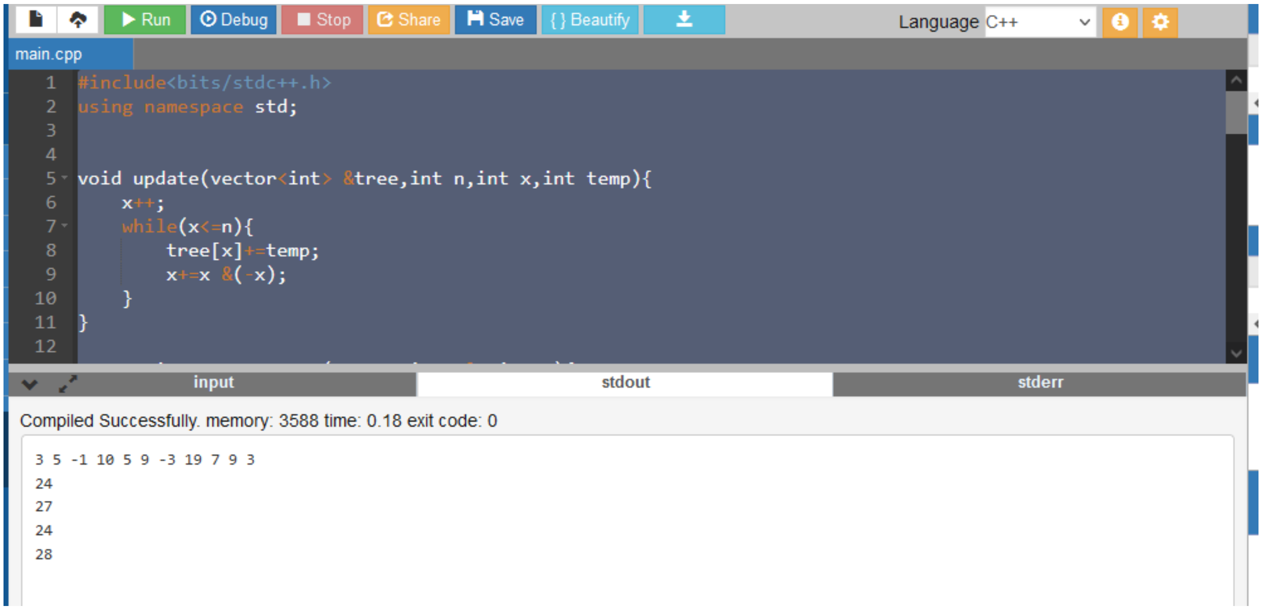
Language C++

```cpp
#include<bits/stdc++.h>
using namespace std;


void update(vector<int> &tree,int n,int x,int temp){
    x++;
    while(x<=n){
        tree[x]+=temp;
        x+=x &(-x);
    }
}

```

Command line arguments:

Standard Input: ○ Interactive Console ● Text

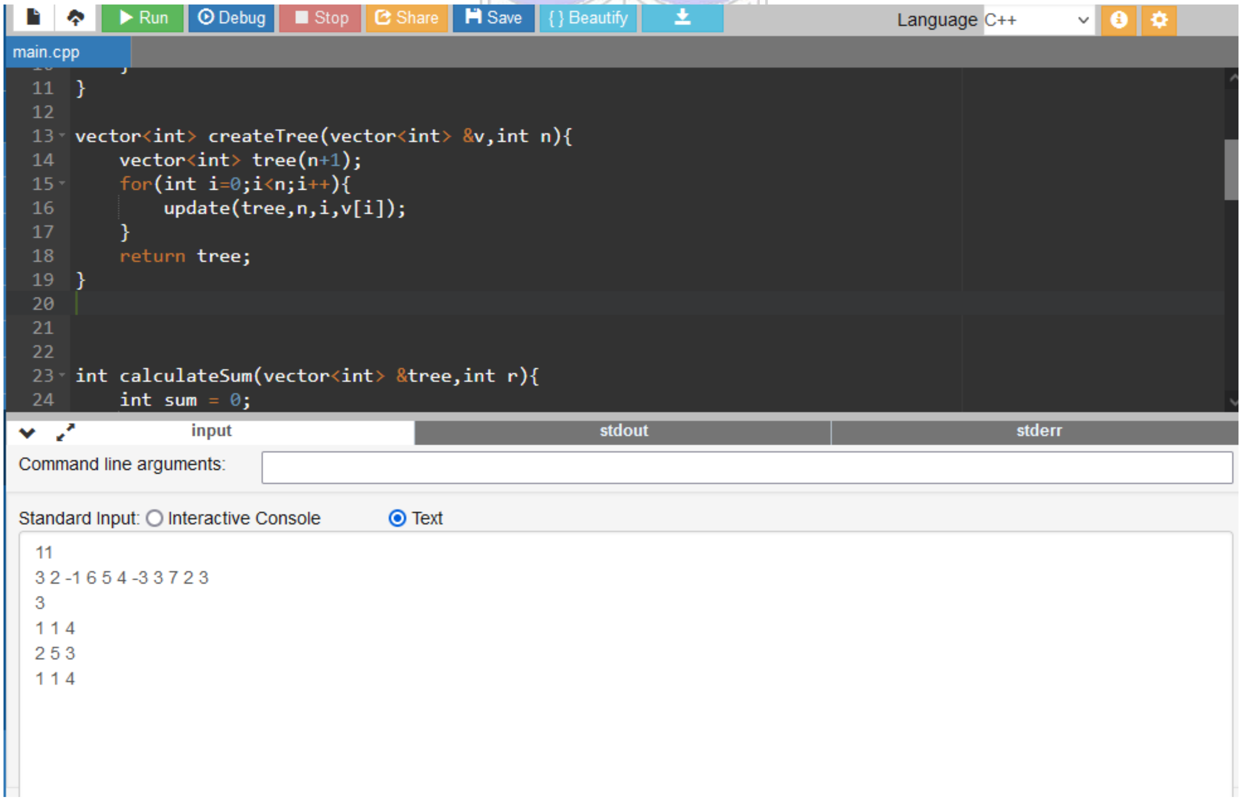
```
11
3 2 -1 6 5 4 -3 3 7 2 3
5
1 3 10
2 5 3
1 3 10
1 2 9
1 1 10
```



```cpp
#include<bits/stdc++.h>
using namespace std;


void update(vector<int> &tree,int n,int x,int temp){
    x++;
    while(x<=n){
        tree[x]+=temp;
        x+=x &(-x);
    }
}
```

Compiled Successfully. memory: 3588 time: 0.18 exit code: 0

```
3 5 -1 10 5 9 -3 19 7 9 3
24
27
24
28
```

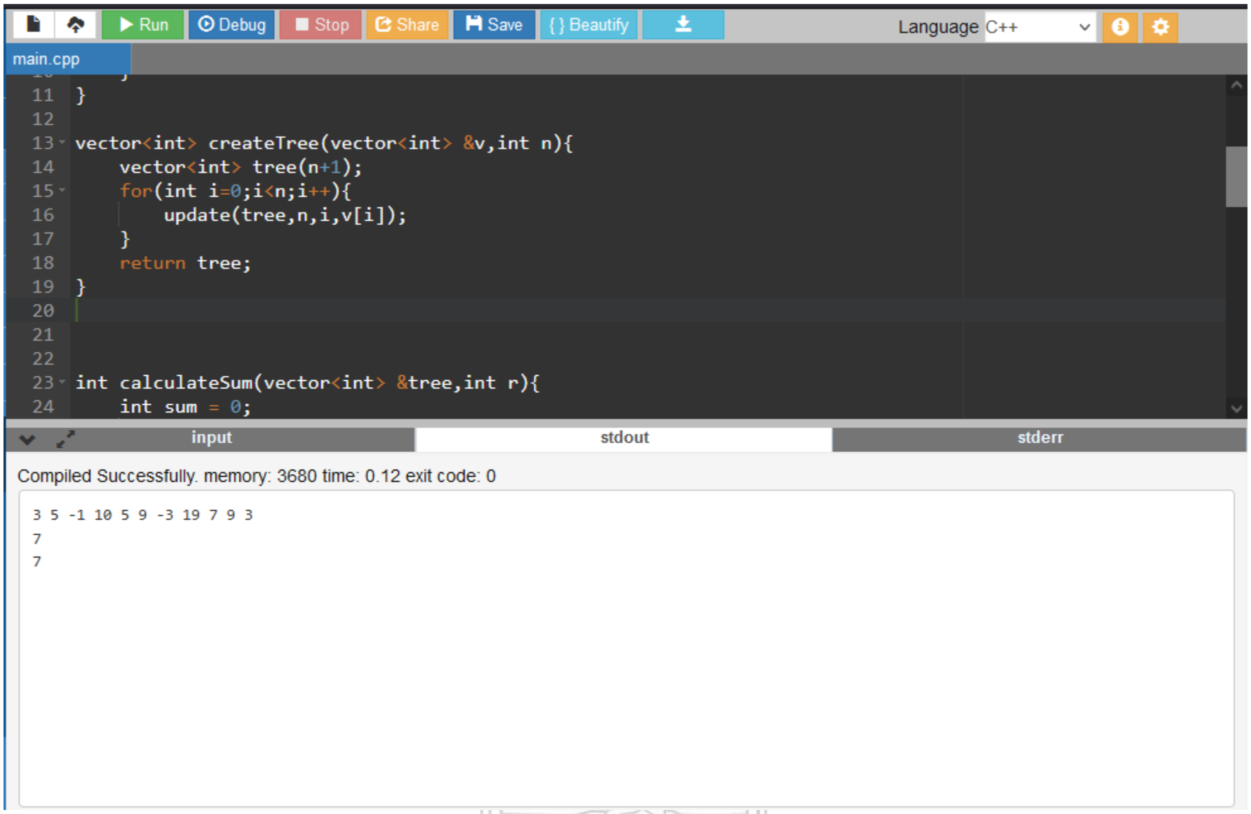


```cpp
}

vector<int> createTree(vector<int> &v,int n){
    vector<int> tree(n+1);
    for(int i=0;i<n;i++){
        update(tree,n,i,v[i]);
    }
    return tree;
}



int calculateSum(vector<int> &tree,int r){
    int sum = 0;
```

Command line arguments:

Standard Input: ○ Interactive Console ● Text

```
11
3 2 -1 6 5 4 -3 3 7 2 3
3
1 1 4
2 5 3
1 1 4
```



```
}

vector<int> createTree(vector<int> &v,int n){
    vector<int> tree(n+1);
    for(int i=0;i<n;i++){
        update(tree,n,i,v[i]);
    }
    return tree;
}



int calculateSum(vector<int> &tree,int r){
    int sum = 0;
```

Compiled Successfully. memory: 3680 time: 0.12 exit code: 0

```
3 5 -1 10 5 9 -3 19 7 9 3
7
7
```

Outcomes:

CO3 . Understand the Graphs, related algorithms, efficient implementation of those algorithms and applications

Conclusion: (Conclusion to be based on the objectives and outcomes achieved)

Understood the concept of fenwick tree and implemented the same. thus observing that we are able to query and update our answers in O(logN) time complexity thus saving time when we want to find answers in a given range.

References:

1. https://www.hackerearth.com/practice/data-structures/advanced-data-ructures/segment-trees/tutorial/

   https://cp-algorithms.com/data_structures/segment_tree.html