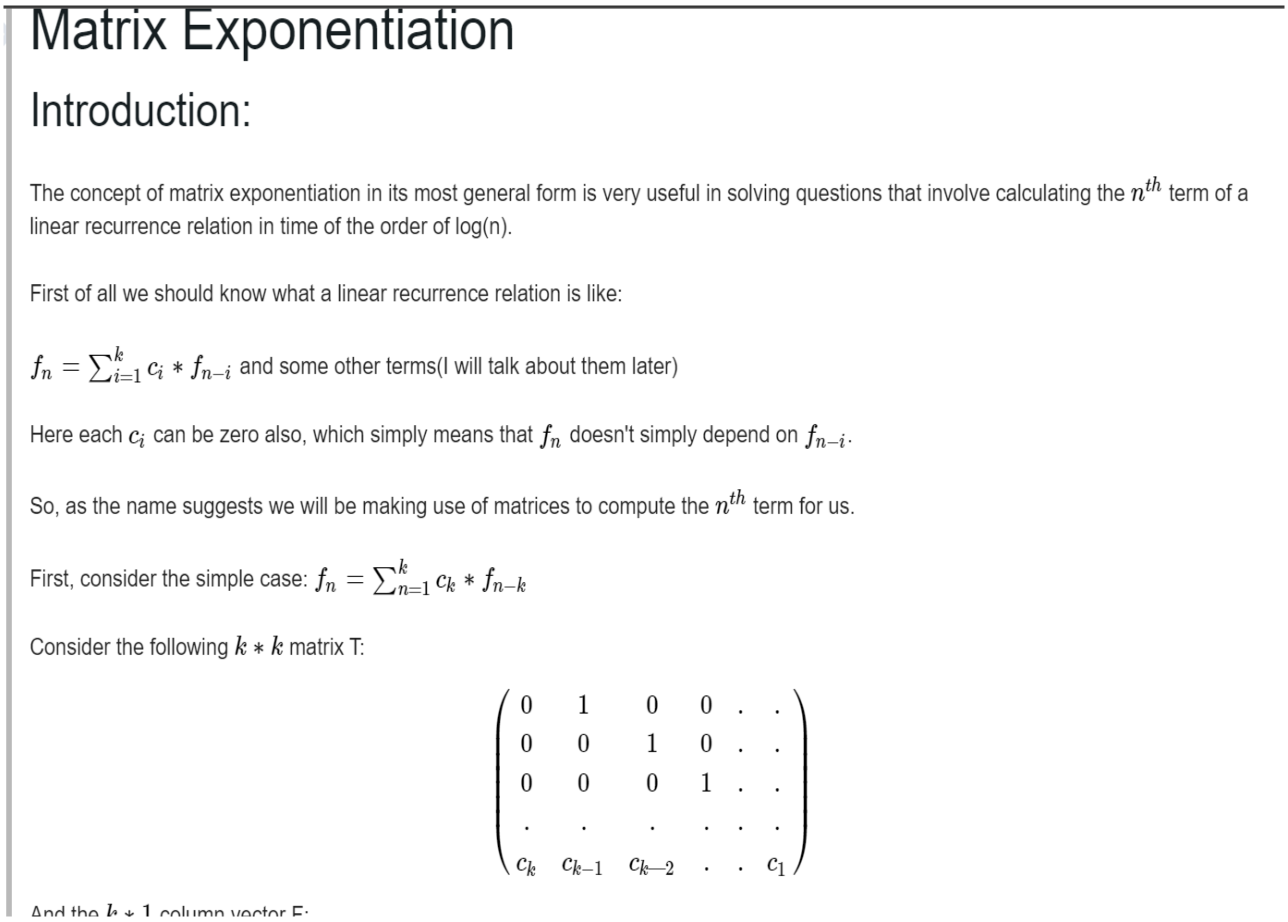
STRIVER CP SHEET

EXTRA TOPICS THAT I DON'T KNOW

1) Matrix exponentiation



Matrix Exponentiation

Introduction:

The concept of matrix exponentiation in its most general form is very useful in solving questions that involve calculating the n^{th} term of a linear recurrence relation in time of the order of log(n).

First of all we should know what a linear recurrence relation is like:

$f_n = \sum_{i=1}^{k} c_i * f_{n-i}$ and some other terms(I will talk about them later)

Here each c_i can be zero also, which simply means that f_n doesn't simply depend on f_{n-i}.

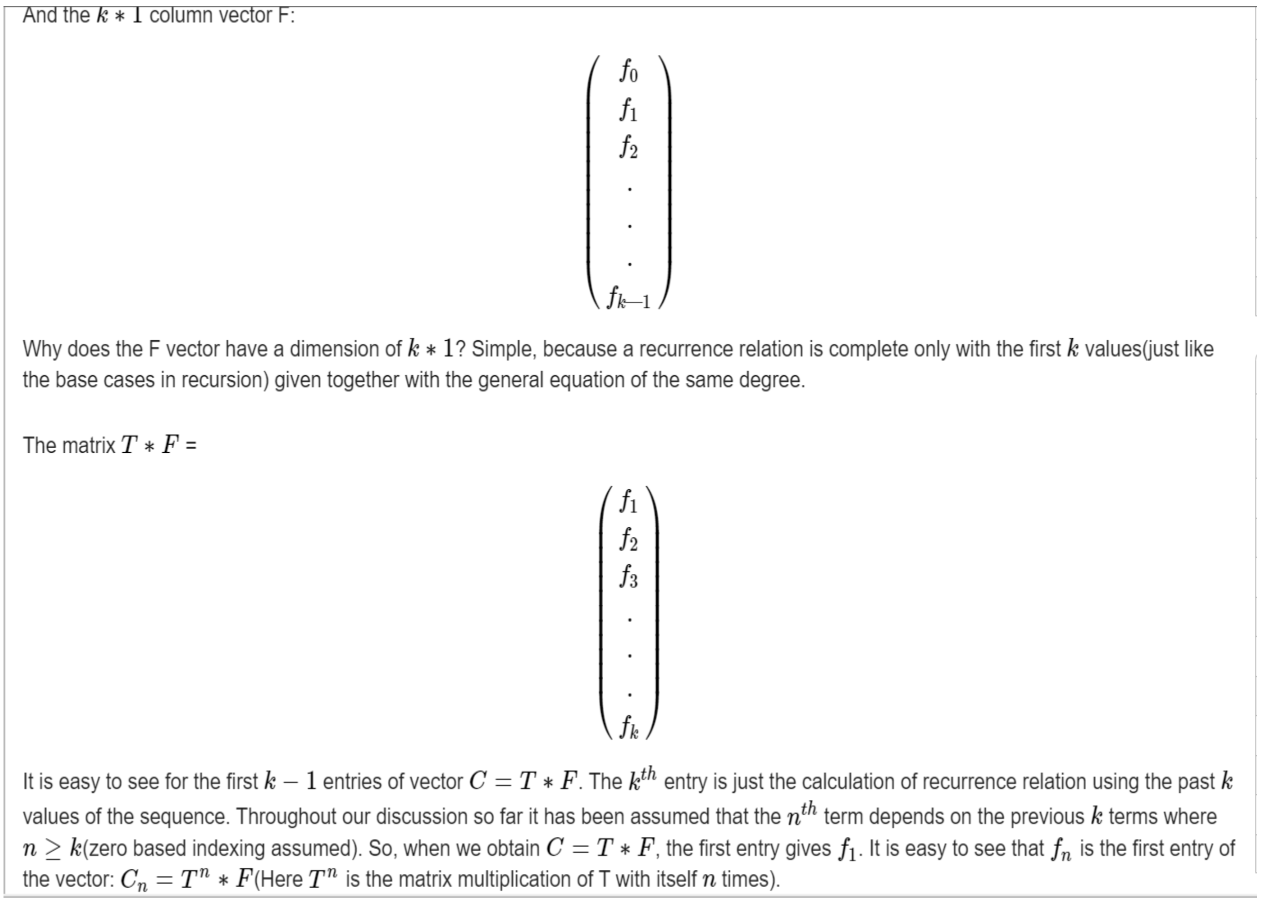
So, as the name suggests we will be making use of matrices to compute the n^{th} term for us.

First, consider the simple case: $f_n = \sum_{n=1}^{k} c_k * f_{n-k}$

Consider the following $k * k$ matrix T:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & . & . \\ 0 & 0 & 1 & 0 & . & . \\ 0 & 0 & 0 & 1 & . & . \\ . & . & . & . & . & . \\ c_k & c_{k-1} & c_{k-2} & . & . & c_1 \end{pmatrix}$$

And the $k * 1$ column vector F:



And the $k * 1$ column vector F:

$$\begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ . \\ . \\ . \\ f_{k-1} \end{pmatrix}$$

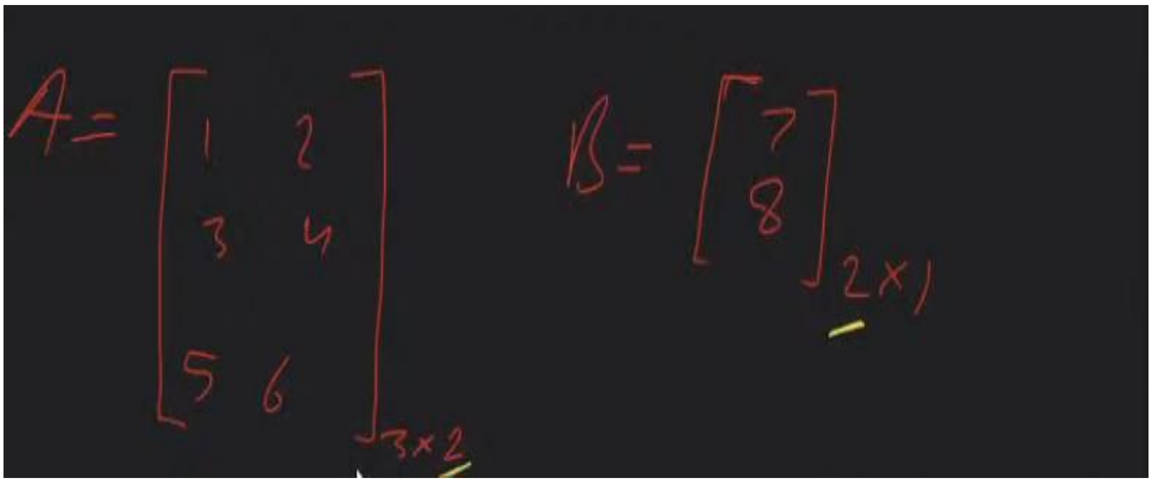
Why does the F vector have a dimension of $k * 1$? Simple, because a recurrence relation is complete only with the first k values(just like the base cases in recursion) given together with the general equation of the same degree.

The matrix $T * F$ =

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ . \\ . \\ . \\ f_k \end{pmatrix}$$

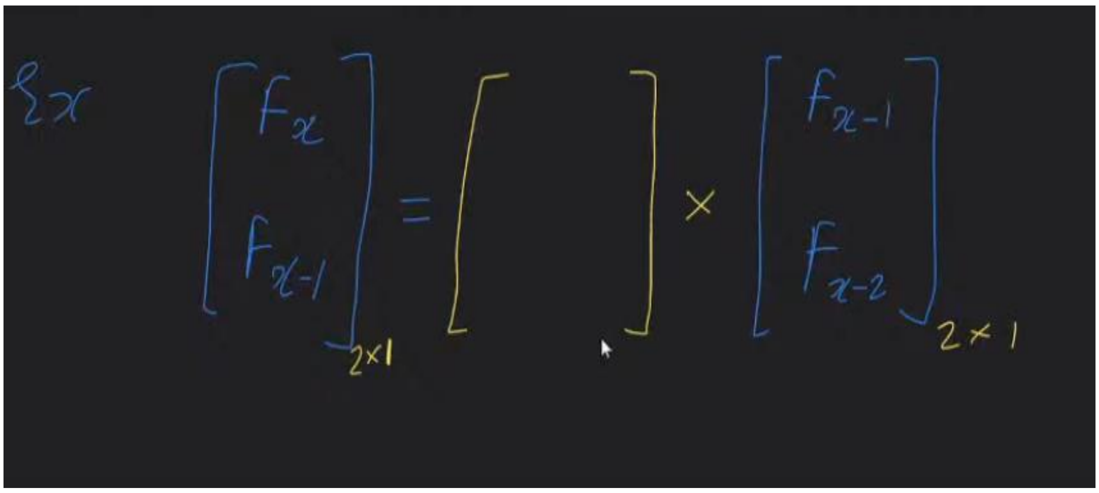
It is easy to see for the first $k-1$ entries of vector $C = T * F$. The k^{th} entry is just the calculation of recurrence relation using the past k values of the sequence. Throughout our discussion so far it has been assumed that the n^{th} term depends on the previous k terms where $n \geq k$(zero based indexing assumed). So, when we obtain $C = T * F$, the first entry gives f_1. It is easy to see that f_n is the first entry of the vector: $C_n = T^n * F$(Here T^n is the matrix multiplication of T with itself n times).

Basically, we multiply 2 matrices of desired size.

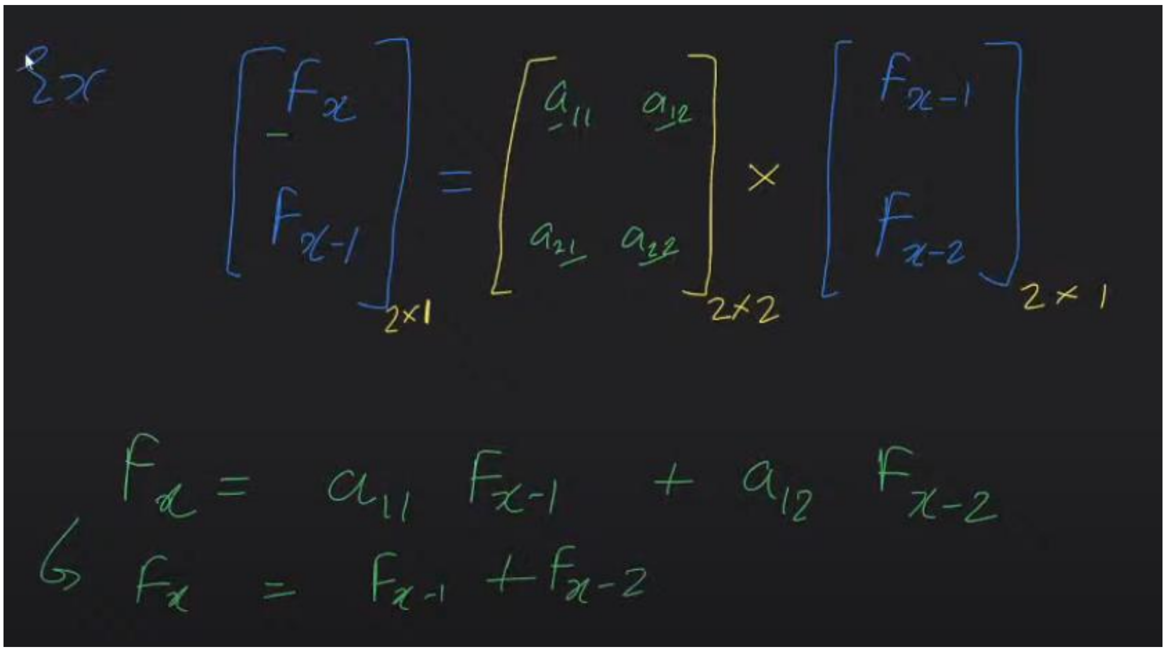

$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}_{3\times 2}$
$B = \begin{bmatrix} 7 \\ 8 \end{bmatrix}_{2\times 1}$

As we saw there is a matrix A of 3x2 and matrix B of 2x1 as their no. of columns and rows respectively are equal then we can easily multiply without any ease.

You won’t get it without solving any questions, so let's solve a reputed question of Fibonacci no.s:-

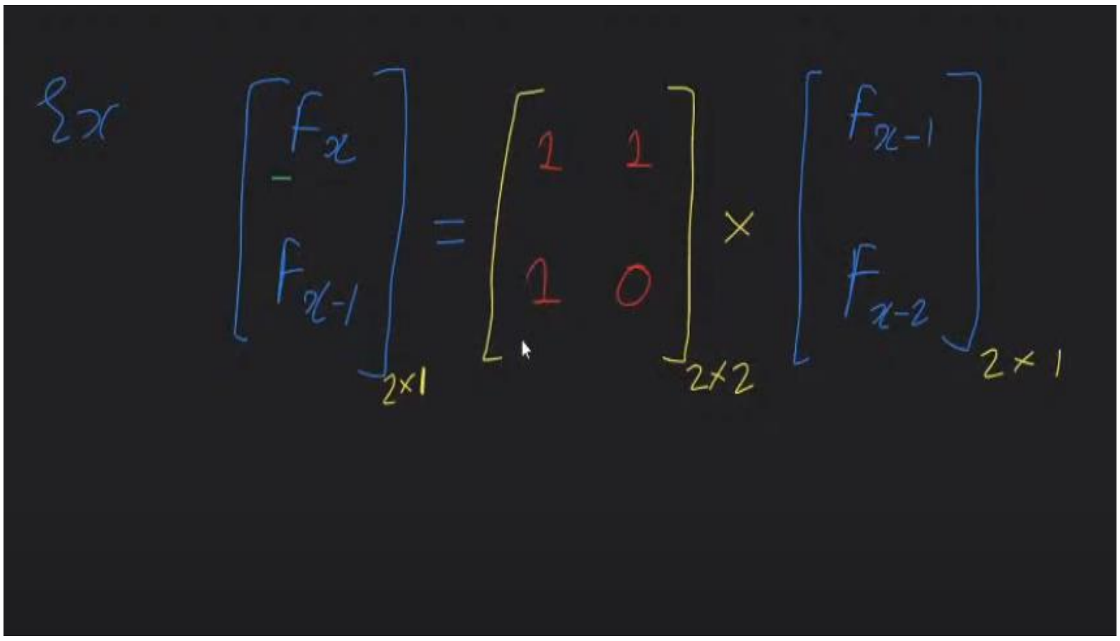

Ex
$\begin{bmatrix} F_x \\ F_{x-1} \end{bmatrix}_{2\times1} = \begin{bmatrix} & \\ & \end{bmatrix} \times \begin{bmatrix} F_{x-1} \\ F_{x-2} \end{bmatrix}_{2\times1}$

First, we have designed the matrix that we are going to solve, and we have taken a matrix of size 2x1 because, in Fibonacci, we need the sum of two no.s to get the answer, now the size of the yellow matrix is 2x2, so lets find the values of that matrix [always find the value of yellow matrix on your own].


Ex
$\begin{bmatrix} F_x \\ F_{x-1} \end{bmatrix}_{2\times1} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}_{2\times2} \times \begin{bmatrix} F_{x-1} \\ F_{x-2} \end{bmatrix}_{2\times1}$
$F_x = a_{11} F_{x-1} + a_{12} F_{x-2}$
$\hookrightarrow F_x = F_{x-1} + F_{x-2}$

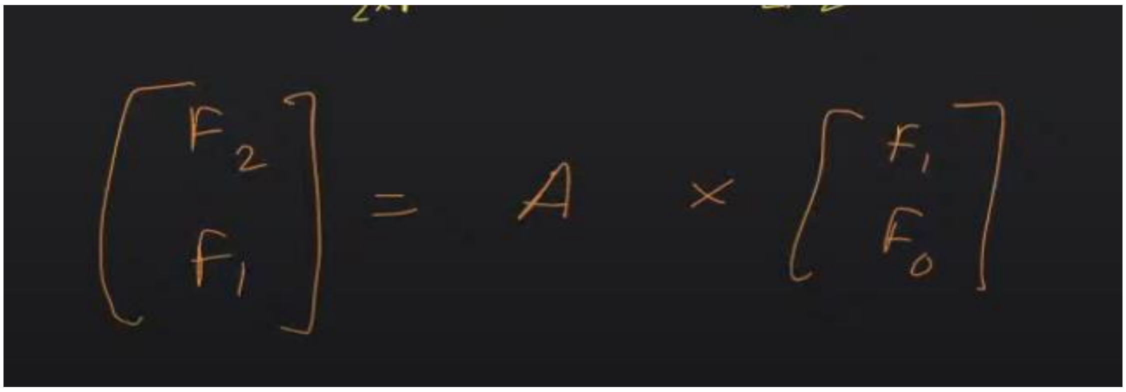
By knowing the Fibonacci relation we can define the values of a11 and a12 which are nothing but 1 and 1 respectively.

And for fx-1 the value itself is present in another matrix which is fx-1 so a21=1 and a22 =0 hence we got our yellow matrix.



$$\text{Ex} \quad \begin{bmatrix} F_x \\ F_{x-1} \end{bmatrix}_{2\times 1} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}_{2\times 2} \times \begin{bmatrix} F_{x-1} \\ F_{x-2} \end{bmatrix}_{2\times 1}$$

Now, we are forming up the general equation to get the answer.



$$\begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = A \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

$$\begin{bmatrix} f_3 \\ f_2 \end{bmatrix} = A \times \begin{bmatrix} F_2 \\ F_1 \end{bmatrix}$$

$$\begin{bmatrix} F_3 \\ F_2 \end{bmatrix} = A \times A \times \begin{bmatrix} f_1 \\ F_0 \end{bmatrix}$$

So finally we get this eqn:-

$$\begin{bmatrix} \underline{F_{n+1}} \\ \underline{F_n} \end{bmatrix} = \underline{A^n} \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

Now, we have to figure out A^n which will be calculated by :-

$$\underline{O(\log(n))}$$

$$\underline{A^n} \longrightarrow \underline{A^{\frac{n}{2}}} \times \underline{A^{\frac{n}{2}}} \quad n \text{ even}$$

$$\hookrightarrow A^{\frac{n}{2}} \times A^{\frac{n}{2}} \times A \quad n \text{ odd}$$

And this is the following code we are following:-

```
#include <iostream>
#include <vector>

using namespace std;

// Define a matrix multiplication function
vector<vector<long long>> multiply(const
vector<vector<long long>>& A, const
vector<vector<long long>>& B, int mod) {
    int size = A.size();
    vector<vector<long long>> result(size,
vector<long long>(size, 0));

    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            for (int k = 0; k < size; ++k) {
                result[i][j] = (result[i][j] + (A[i][k] *
B[k][j]) % mod) % mod;
            }
        }
    }
    return result;
}

// Define a matrix exponentiation function
vector<vector<long long>> matrixPower(const
vector<vector<long long>>& base, long long exp,
int mod) {
    int size = base.size();
    vector<vector<long long>> result(size,
vector<long long>(size, 0));

    // Initialize result matrix as identity matrix
    for (int i = 0; i < size; ++i) {
        result[i][i] = 1;
    }

    while (exp > 0) {
        if (exp % 2 == 1) {
            result = multiply(result, base, mod);
        }
        base = multiply(base, base, mod);
        exp /= 2;
    }

    return result;
}

// Function to calculate nth Fibonacci number
using matrix exponentiation
long long fibonacci(int n, int mod) {
    if (n == 0) return 0;

    vector<vector<long long>> base = {{1, 1}, {1, 0}};
    vector<vector<long long>> result =
matrixPower(base, n - 1, mod);

    return result[0][0];
}

int main() {
    int n, mod;
    cout << "Enter the value of n: ";
    cin >> n;

    cout << "Enter the modulus value: ";
    cin >> mod;

    long long result = fibonacci(n, mod);
    cout << "The " << n << "th Fibonacci number
modulo " << mod << " is: " << result << endl;

    return 0;
}
```

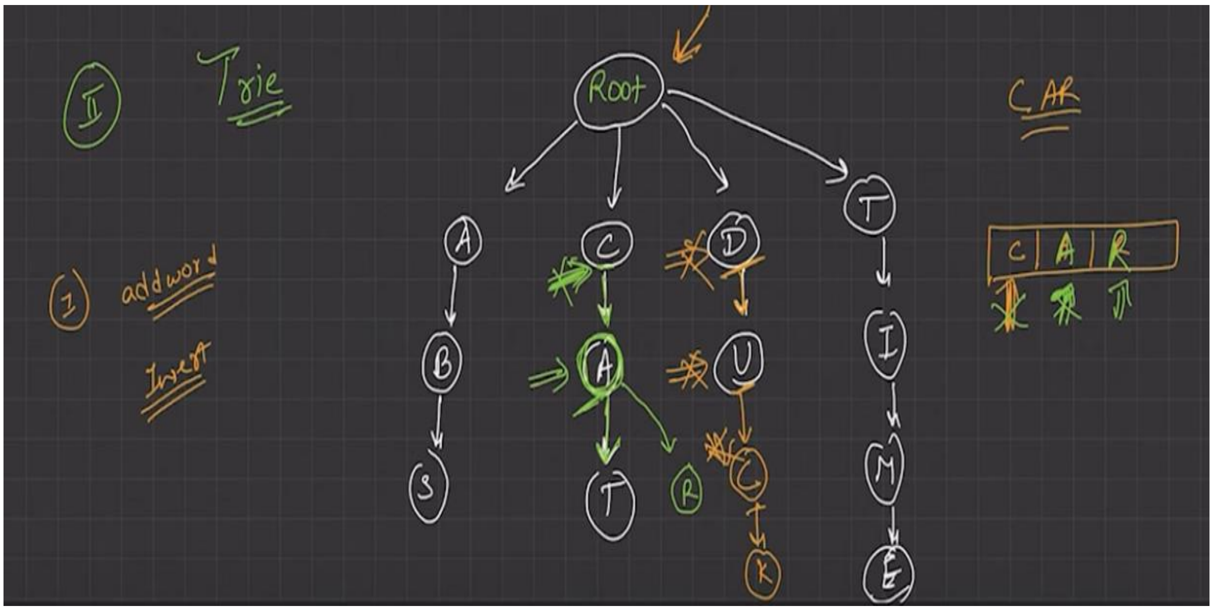
2) TRIE

A trie (pronounced "try") is a tree-like data structure that is used in computer science primarily for storing and searching associative data structures, such as a dynamic set or an associative array where the keys are usually strings. Tries have several use cases in different domains, and one of the most common applications is in the implementation of dictionaries or symbol tables. Here are some reasons why tries are used in data structure and algorithm applications:

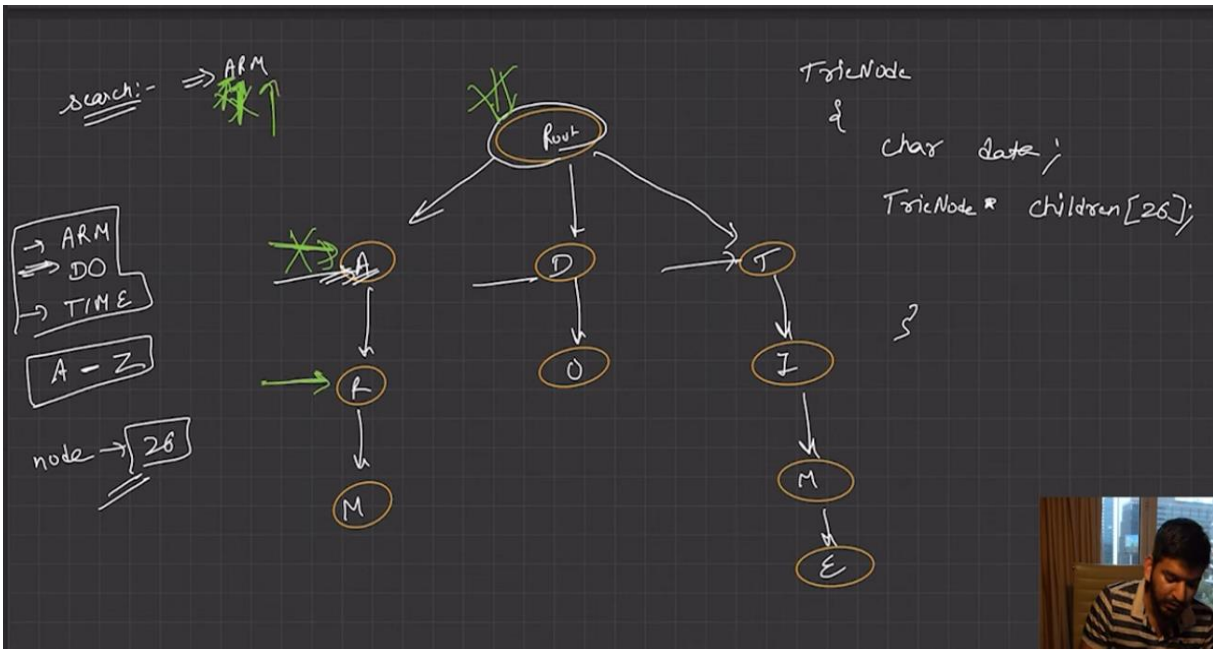
1. **Efficient Prefix Searches:**
   - Tries are particularly efficient when it comes to prefix searches. Searching for words with a common prefix (e.g., autocompletion or spell checking) can be done very quickly using a trie.
2. **Space Efficiency:**
   - Tries can be more space-efficient than alternative data structures, especially when many keys share common prefixes. This is because a trie doesn't store the same prefix multiple times.
3. **Ordered Operations:**
   - Tries can support ordered operations efficiently, making it easy to find the successor or predecessor of a given key. This can be useful in applications like spell checking, where you might want to suggest the next likely word.
4. **Dynamic Set Operations:**
   - Tries allow for dynamic set operations like insertion and deletion with a time complexity proportional to the length of the key. This is advantageous for applications where the set of keys changes frequently.
5. **Memory Efficiency:**
   - Tries are useful in scenarios where memory efficiency is crucial. Tries can be more memory-efficient than hash tables in certain cases, especially when dealing with keys that share common prefixes.
6. **String Matching and Compression:**
   - Tries are used in string matching algorithms, where they can efficiently locate occurrences of a pattern within a body of text. Additionally, tries can be employed in data compression techniques.
7. **Routing Tables:**
   - Tries are used in networking for efficient routing table lookups. IP addresses can be stored in a trie, allowing for quick and efficient routing decisions.
8. **Multi-Key Search:**
   - Tries are well-suited for scenarios where multi-key searches are required. For example, finding all keys with a common prefix.
9. **Text Editors and Compilers:**
   - Tries are used in text editors and compilers for quick keyword searches and lexical analysis.

In summary, tries are versatile data structures with advantages in terms of efficiency for certain types of data and operations. They are particularly well-suited for scenarios involving strings or keys with common prefixes and for applications where quick insertion, deletion, and lookup operations are essential.

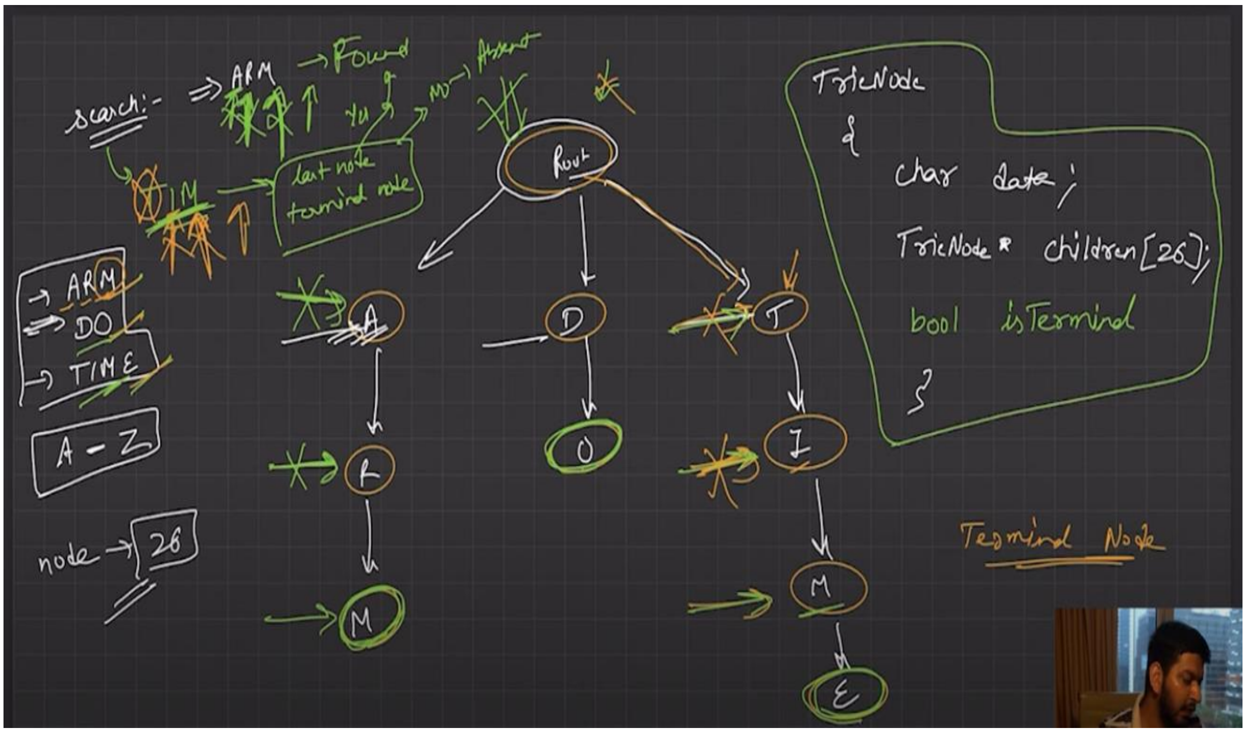
Inserting in existing or new trie:-



Pehle to wo search krega , nhi mila to add krdega,jaise cat me car dhund rha tha , pr R nhi mila , but usne add krdia.



Isme humlog time dhundenge to mil jaega , pr agr TIM dhundenge to wo bhi miljaega lekin wo nhi milna chahie becoz humne word TIME dala h TIM nhi , so case ko handle krne kelie , hr word ke aakhri letter ko terminal node ka naam dia h .



Maanlo TIM milgya , fir ek condition check hogi ki kya jis node me tum ho wo kya koi terminal node h , agr nhi h ,to iska matlb tumhe word jo mila h wo , wo word nhi jisse tu dhundna chah rhe ho .

Implementation:-

First, we have created a tree data structure:-

SQRT decomposition

Square Root Decomposition is a technique used in competitive programming to efficiently handle range queries on an array or any other data structure. It's particularly useful when dealing with problems that involve multiple queries of two types: updates and range queries.

Here's a basic overview of how Square Root Decomposition works and why it's used:

Basic Idea:

1. **Divide into Blocks:** The array is divided into blocks, each containing a certain number of elements. The size of these blocks is usually chosen to be around the square root of the total number of elements in the array.
2. **Preprocessing:** Precompute some information for each block to speed up the query process. This step is often done during the initialization of the data structure.
3. **Handle Queries:**
    - **Update Queries:** Modify the array in a standard way.
    - **Range Queries:** Process the query by considering the blocks in a specific range and using precomputed information for those blocks.

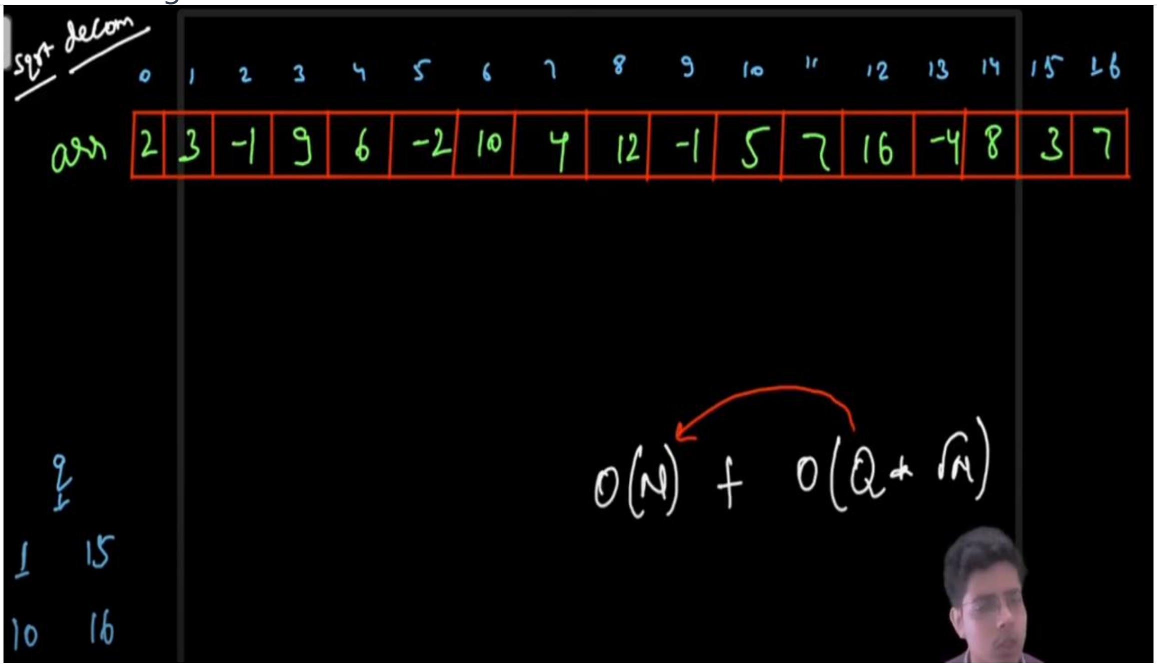
Why it is Used:

1. **Efficiency:** Square Root Decomposition allows you to reduce the time complexity of range queries from linear to sub-linear, making it more efficient.
2. **Trade-off:** It provides a trade-off between time and space complexity. While there is some extra space needed for precomputed information, the overall time complexity for range queries is significantly improved.
3. **Simplicity:** It's a relatively simple technique that can be implemented with a few lines of code, making it suitable for competitive programming where concise and efficient solutions are crucial.
4. **Versatility:** Square Root Decomposition is not limited to arrays; it can be adapted for other data structures like trees, graphs, etc., depending on the problem requirements.

Example Problem:

A common example where Square Root Decomposition is applied is in solving problems related to finding the sum or minimum/maximum in a range of an array, where the array is subject to both updates and queries.

Keep in mind that while Square Root Decomposition is a powerful technique, it may not be the best fit for every problem. It's essential to understand the problem requirements and constraints to choose the most appropriate algorithm or data structure for a given scenario.

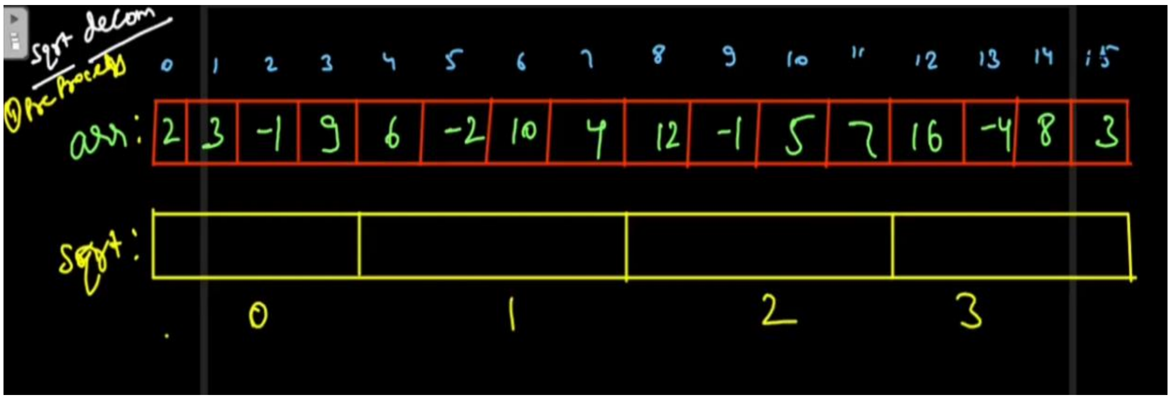


Here we have to do some precomputation of O(n) time complexity and another function which is done in O((q(no. of queries))*(square_root(n))).

Where n is the length of the array.

Now, what we do in pre-processing:-

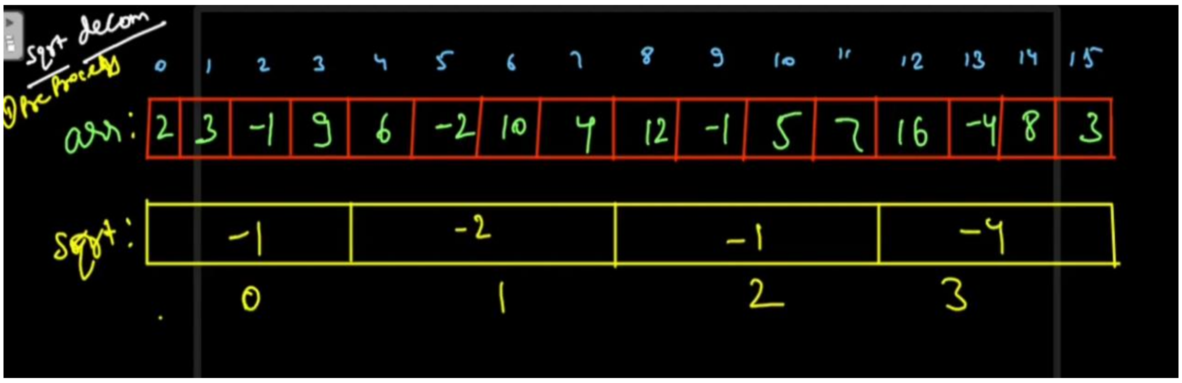
Here we create an array called SQRT here the length will be size ceil(root n).



Here we have created a sqrt array, and in each index/box, we will store the desired data of the corresponding root(n) elements of the array.

Ex here root(n)=4, so arr is divided into 4 compartments like(2,3,-1,9) and so on, and from these compartments we will store the data according to the conditions.

Like in youtube question, is asked to find the minimum, so we store the minimum ,

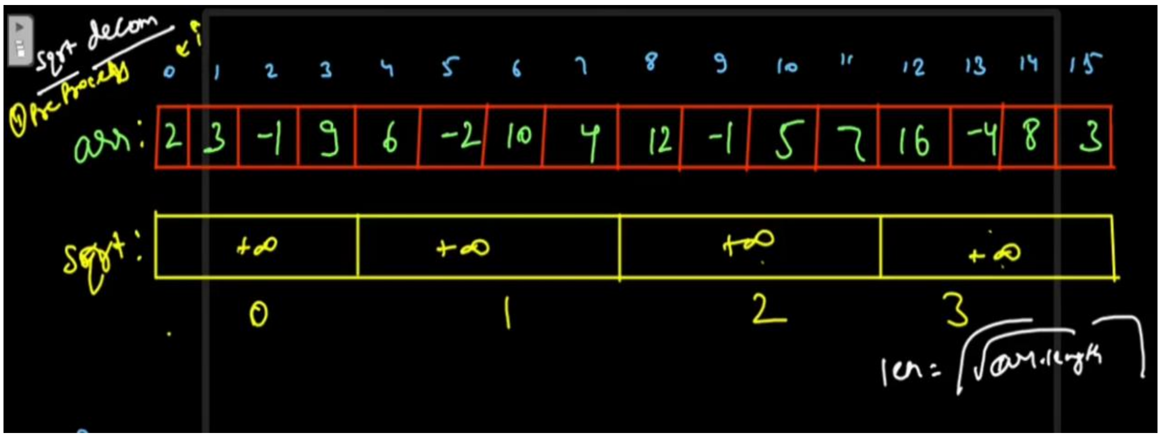


Here we stored the minimums.

Now see, mapping of indexes has been done in such a way that is, the index of array/ceil(root(n)),

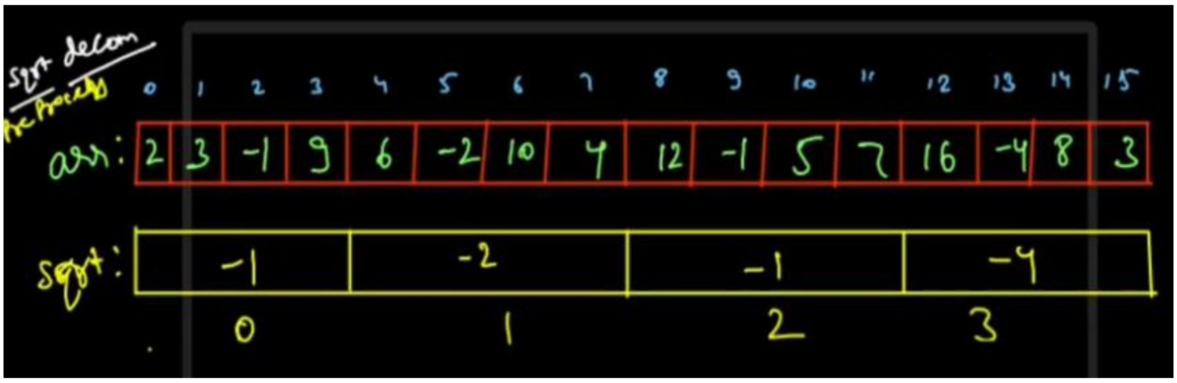
Like 0/4=0 which is the 0th index of array is corresponds to the 0th index of sqrt array.

But initially, in the sqrt array, we have assigned the values of infinity,

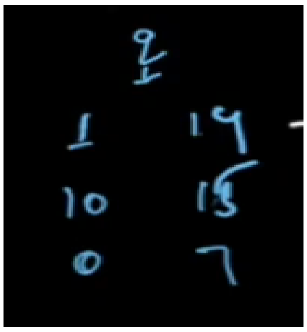


Now, we start traversing through the main array (arr), and then we start updating the values of sqrt arrays, like sqrt[i(index of main array)/ceil(root(n))]=min(sqrt[i/ceil(root(n))],arr[i]).

So finally we get this:-



Now, we provided with certain queries, for ex:-



1 and 14 tells that , we have to find the minimums between the 1st and 14th elements.

[how do we know that which element of the main array is corresponding to which element of sqrt array , and it is defined by {i(index of main element)%(ceil(root(n))}.

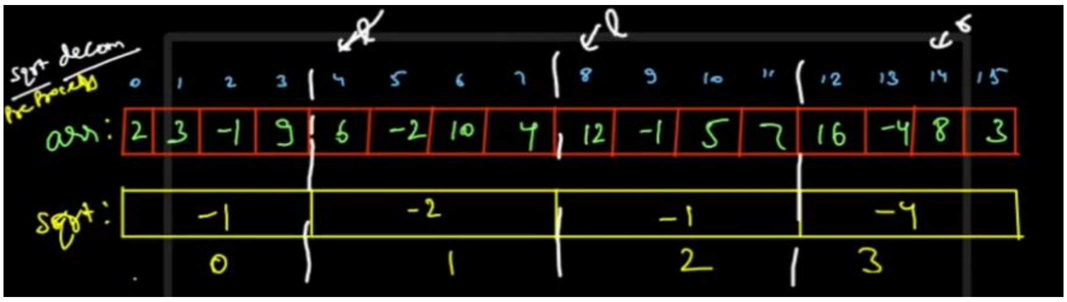
Now see, we have to calculate for 1 to 14 , do why would we calculate linearly for 4 to 11 because their minimum we have already calculated in sqrt array ,we have to linearly solve for partial components like 1 and 4 in this example.

And we know when it is a partial component or not by this condition,

$$l + len - 1 <= r$$

len=ceil(root(n)),l=present index in main array while traversing, r=limit of the query , for 14 in this ex;

And after using sqrt index , please update the values of l , where l+=len.

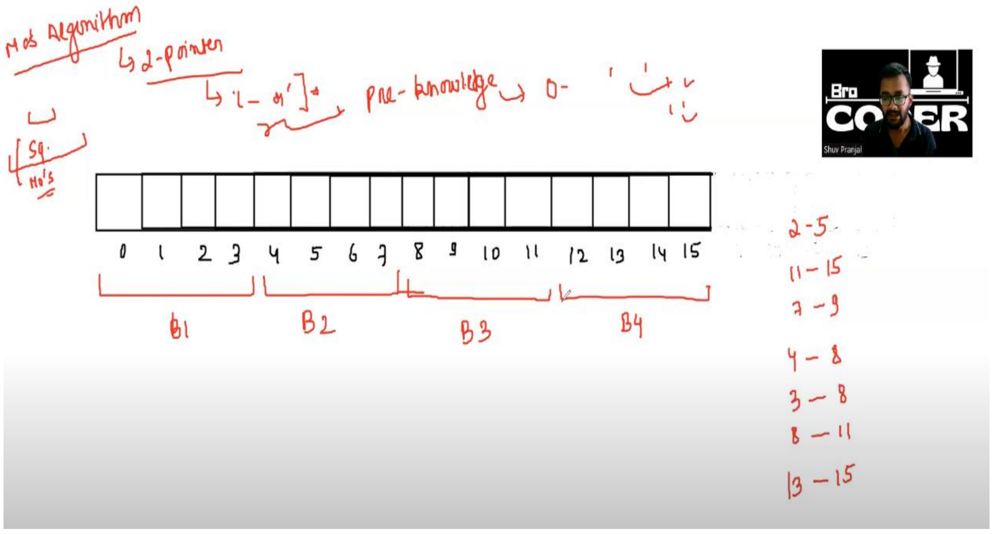


Further iterating, we get our minimum for the 1st query. By simply following this procedure we can solve for all queries.

3) ***MO's algorithm (base on sqrt decomposition):-***

It can also be termed an advanced 2-pointer method.

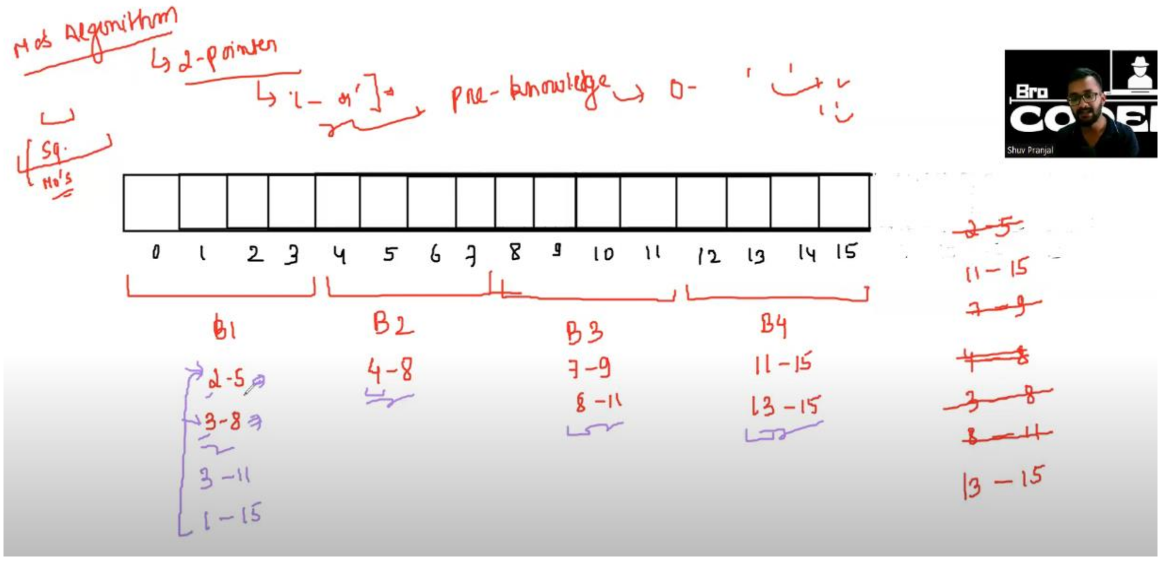
We used this algorithm because sqrt decompositions are not useful for questions like finding the no. of occurrences in ranges queries .



Here we also create sqrt array for storing the elements.

Now, divide queries according the indexes of the elements , for ex

(2-5) will lie in box 1 as 2 lies in box 1.



In each box, range queries are arranged in ascending order such that whichever range has more higher limit then It would be played at last.

Now, we have created two pointers named l and r,

l corresponds to the lower limit of the range and r corresponds to the higher limit of that given range.

The left pointer (l) only moves in a given box and the right pointer (r) moves from (r) to (n(length of the array)).

Here, left and right pointers will also move according to the queries that's what l will move the given to box, and r can move throughout the array.

For each box, (r) performs (n) operations and (l) does, q(no. of queries)*(root(n)) operations, and we also sorted the queries which add (q*log(q)) time .

The overall time complexity is given as:-

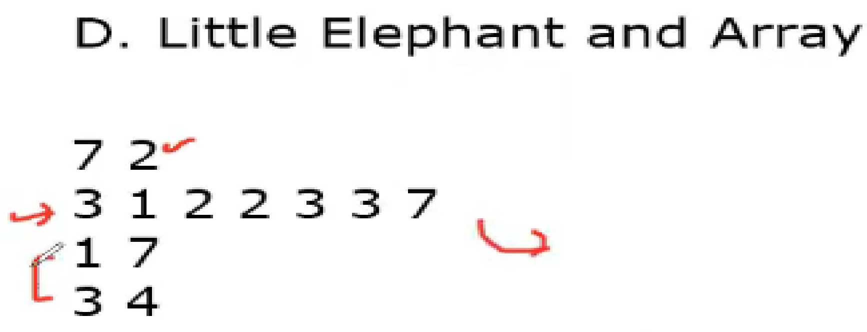
$$TC: \quad O(\underbrace{N\sqrt{N} + Q\sqrt{N}}_{\downarrow} + \underbrace{\boxed{Q \log Q}}_{\downarrow})$$

$$\rightarrow TC: \quad O(\sqrt{N}(N + Q))$$

$$\rightarrow SC: \quad O(\sqrt{N})$$

Now we are solving a question:-

Link-- https://codeforces.com/contest/220/problem/B



D. Little Elephant and Array

```
7 2
3 1 2 2 3 3 7
1 7
3 4
```

As we are sorting the queries we have to store the sequence of queries being asked in order to provide the solution.

And this is the solution:-

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

const int MXN = 1e5 + 5, BLOCK_SIZE = 316;

struct Query {
   int l, r, idx;
};
   bool cmp( Query y, Query x) {
      // Current query x is being compared with
other query y
      int x_block = x.l / BLOCK_SIZE;
      int y_block = y.l / BLOCK_SIZE;

      // If x and y both lie in the same block, sort in
non decreasing order of endpoint
```

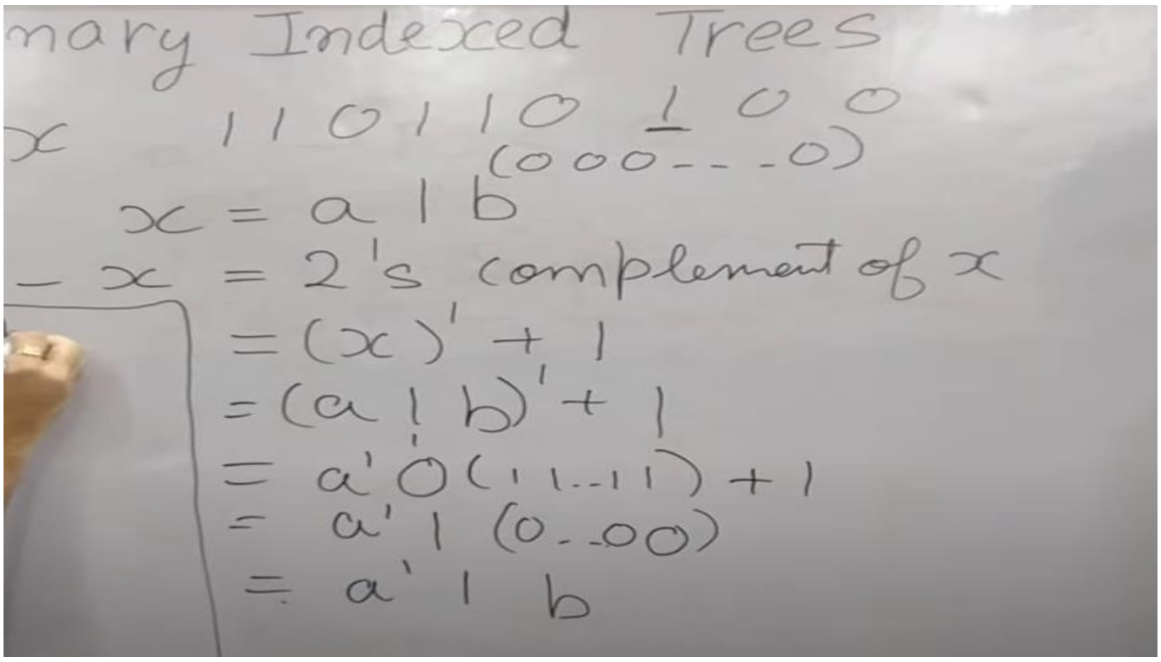
```
        if (x_block == y_block)
            return x.r < y.r;

        // x and y lie in different blocks
        return x_block < y_block;
    }


int A[MXN], final_ans[MXN], freq[MXN];
int cur_ans;

void operate(int idx, int delta) {
    int x = A[idx];
    if (x >= MXN) return;

    if (freq[x] == x) cur_ans--;
    freq[x] += delta;
    if (freq[x] == x) cur_ans++;
}

signed main() {
    int N, M;
    cin >> N >> M;

    for (int i = 0; i < N; i++)
        cin >> A[i];

    vector<Query> queries(M);
    for (int i = 0; i < M; i++) {
        cin >> queries[i].l >> queries[i].r;
        queries[i].l--, queries[i].r--; // 0 based
indexing

        queries[i].idx = i;
    }

    sort(queries.begin(), queries.end(),cmp);

    int i = 0, j = -1;
    for ( auto &q : queries) {
        int l=q.l,r=q.r,idx=q.idx;
        while (j < r) operate(++j, 1);
        while (i > l) operate(--i, 1);

        while (i < l) operate(i++, -1);
        while (j > r) operate(j--, -1);

        final_ans[idx] = cur_ans;
    }

    for (int i = 0; i < M; i++)
        cout << final_ans[i] << "\n";

}
```

3)Fenwick tree:-

If you're referring to Binary Indexed Trees (BIT), also known as Fenwick Trees, I can provide some information.

Binary Indexed Trees is a data structure that allows efficient updates and queries of prefix sums in an array. They are particularly useful when you need to perform cumulative operations on a dynamic array, such as finding the sum of elements in a range or updating individual elements. Binary Indexed Trees achieve this with a relatively low time complexity. The key idea behind Binary Indexed Trees is to represent the array as a binary tree, where each node stores the sum of some elements. This structure enables efficient updates and queries. BIT supports operations like updating the value of an element and finding the prefix sum of a range of elements, both in logarithmic time.

Here, first of all, we need to find the rightmost set bit of a given no. :-



nary Indexed Trees

$x \quad 1101110\underline{1}00$

$(000 \ldots 0)$

$x = a1b$

$-x = 2\text{'s complement of } x$

$= (x)' + 1$

$= (a1b)' + 1$

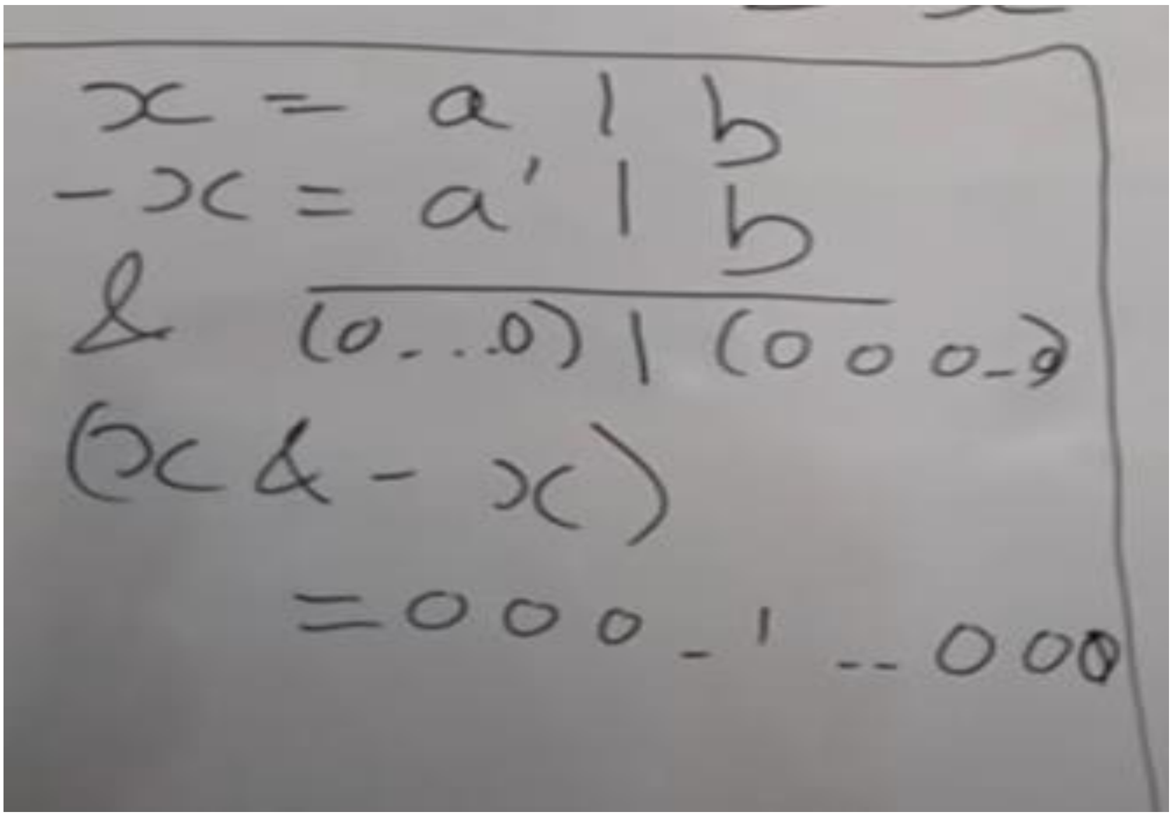
$= a'0(11..11) + 1$

$= a'1(0..00)$

$= a'1b$

Here, we have to find the rightmost set bit, so initially, we have assigned X as (a1b) and calculated 2’s complement of x which is nothing but (-x) .b is initially a binary representation that contains only 0, and we get our -X as (a’1b).

Now, we perform the (and) operation for X and -X so that we get rightmost set bit:-



$x = a\,1\,b$

$-x = a'\,1\,b$

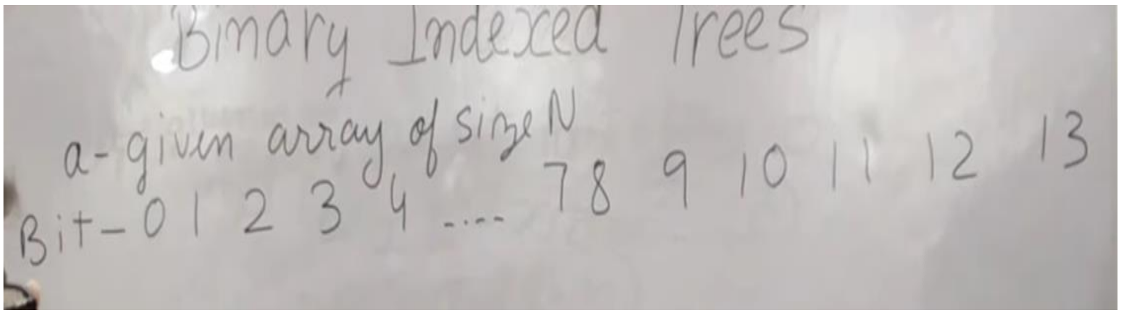
$\& \quad \overline{(0..0)\,1\,(000..0)}$

$(x \& -x)$

$= 000\_1..000$

if we have to remove rightmost set bit, then we will do this:- (X-(X&-X));

Now, we will see how binary index tree store elements:-



Binary Indexed Trees

a- given array of size N

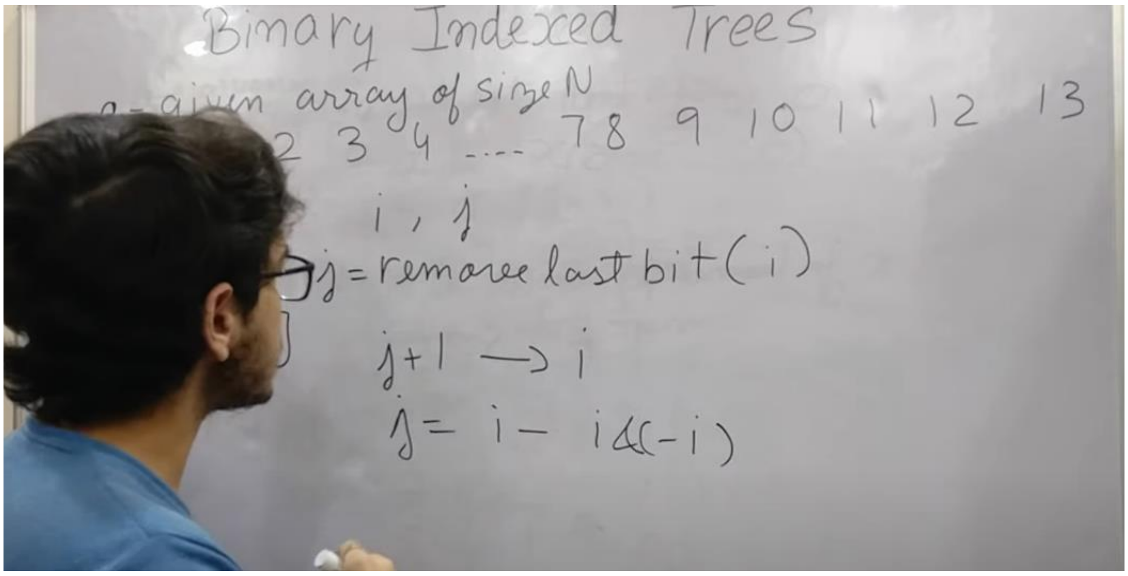
Bit- 0 1 2 3 4 .... 7 8 9 10 11 12 13

Here, every index of a binary index tree(bit) array will store the sum of partial range means the sum of some continuous no.s which is present in the array.

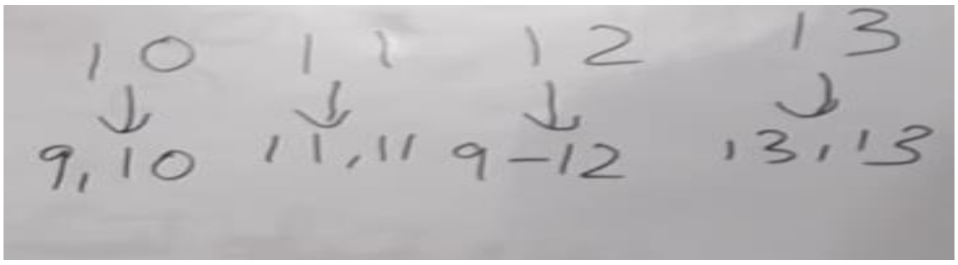
Now, how do we know the range, it will be calculated as:-

Suppose we have two no.s (i) and (j), where (i) is any index of the array and (j) is the number that is equal to the removed last bit of (i).

And, every index of the bit array will store the sum of no.s from (j+1)th index to (i)th index.

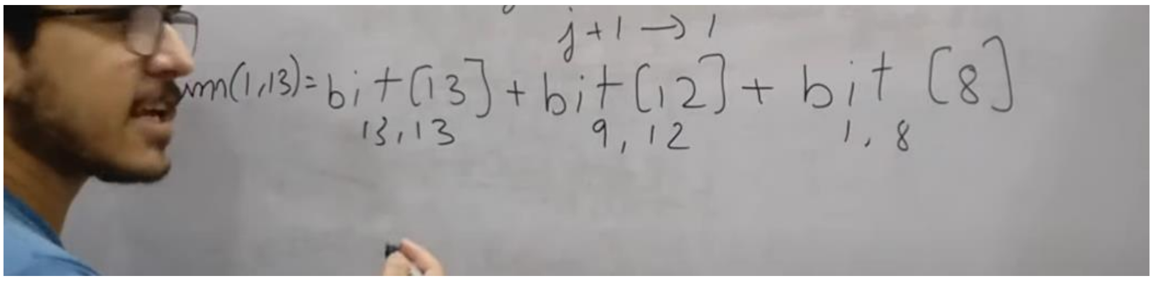


And we have some the partial sum stored by the given index as:-



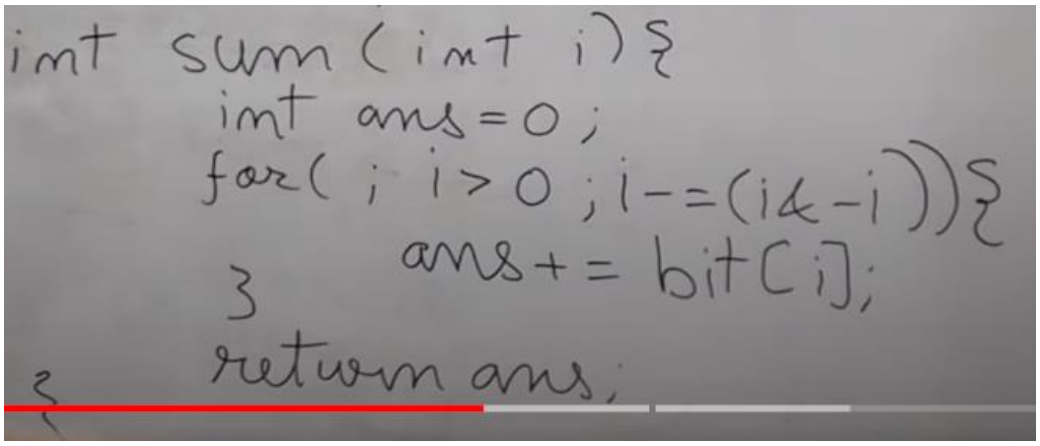
If we are provided with the calculated bit array how would we find the sum of the elements from (1)st to (i)th index of an array.

For example, we have to calculate the sum of 1 to 13 elements, so we will do this:-



Here, first, we added bit [13] then removed (last set bit of 13 and plus 1)and got 12, then we added bit [12], then we (removed last set bit of 12 plus 1) and got 8 then bit [8] added further we (removed last set bit plus 1 and got 1) so we got 1 finally and we have started from 13, so we get the sum of 1st to 13th elements of an array.

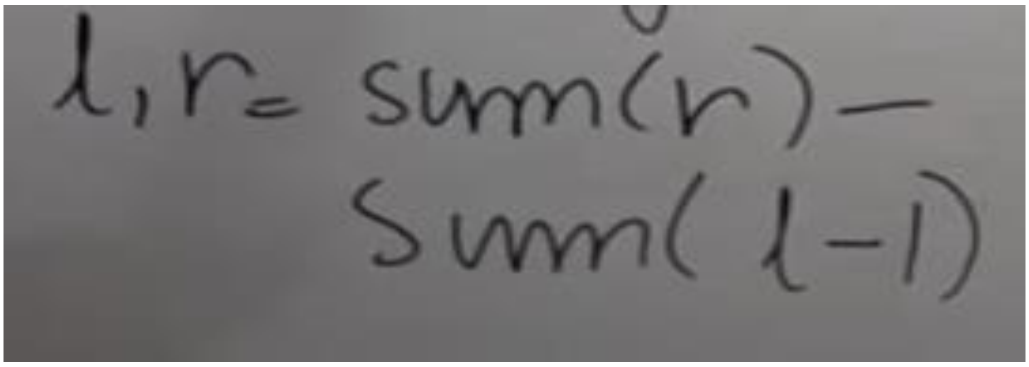
Code for this summation:-



```
int sum(int i){
        int ans=0;
        for( ; i>0 ;i-=(i&-i)){
              ans+= bit[i];
        }
        return ans;
}
```

Here in picture I is the index till where we have to find the sum;

And for any range such as sum of l to r can be calculated as :-



$l, r = \text{sum}(r) - \text{Sum}(l-1)$

Now, we will see how to construct a bit array,

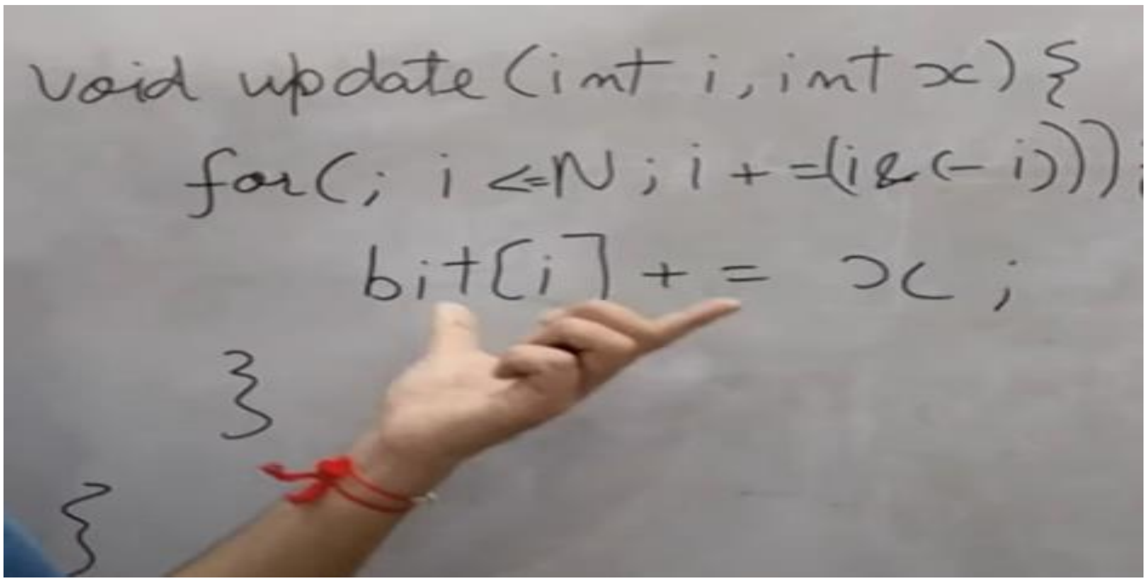
Initially, all values of the bit array will assigned as 0.

Let's say we have to add x on (i)th index, so we will follow this rule:-

1) We will also add x to those indexes in which after removing the last set bit, the (i)th index is coming in their range.
2) Let's say e have (i)th index is given as 13, we will add the last set bit of 13 to 14, so we get 14, now if we see, if we (removed last set bit

of 14 plus 1) we get 13, that’s why we have added last set of 13 to13 so we get to know which numbers are there in which after removing their last set bit , 13 is coming in their range;

And that’s how we have coded :-

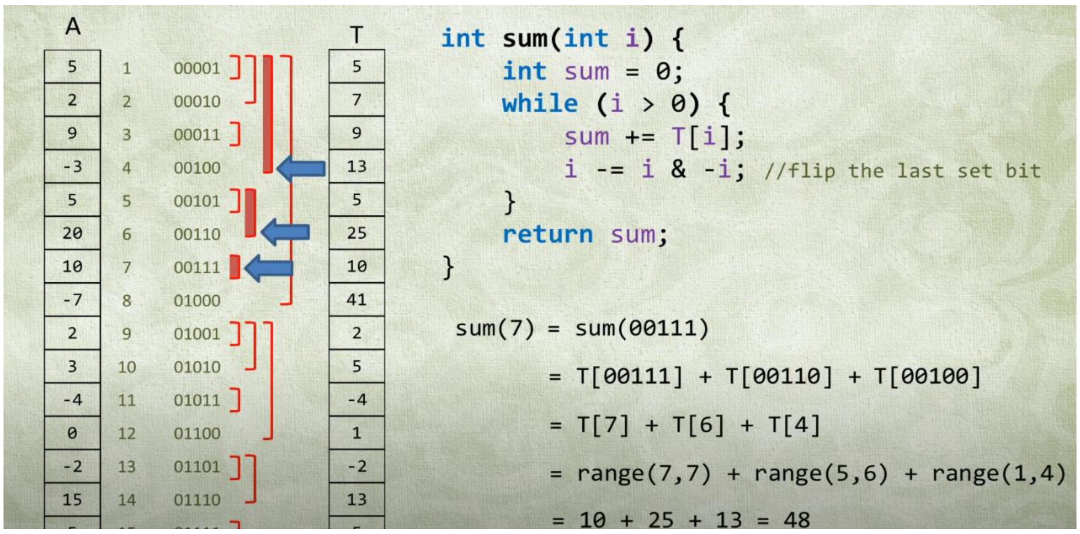


```
void update(int i, int x) {
    for( ; i <= N ; i += (i&(-i)))
        bit[i] += x ;
    }
}
```

It is advisable to keep 1-based indexing in bit array.

Q) find the sum of 1 to 7 in a given array, then we can do this:

Entire fenwick tree code:-



A			T
5	1	00001	5
2	2	00010	7
9	3	00011	9
-3	4	00100	13
5	5	00101	5
20	6	00110	25
10	7	00111	10
-7	8	01000	41
2	9	01001	2
3	10	01010	5
-4	11	01011	-4
0	12	01100	1
-2	13	01101	-2
15	14	01110	13

```
int sum(int i) {
    int sum = 0;
    while (i > 0) {
        sum += T[i];
        i -= i & -i; //flip the last set bit
    }
    return sum;
}
```

```
sum(7) = sum(00111)

       = T[00111] + T[00110] + T[00100]

       = T[7] + T[6] + T[4]

       = range(7,7) + range(5,6) + range(1,4)

       = 10 + 25 + 13 = 48
```

Where T is the BIT array.

entire code for fenwick tree:-

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

class FenwickTree {
private:
    vector<int> BIT;

public:
    FenwickTree(int size) {
        BIT.resize(size + 1, 0);
    }
    // Update the value at index i by delta
    void update(int i, int delta) {
        i++;  // Convert 0-based index to 1-based
index
        while (i < BIT.size()) {
            BIT[i] += delta;
            i += (i & -i);  // Move to the parent in the
binary tree
        }
    }

    // Construct Fenwick Tree from an input array
    void constructTree(const vector<int>& input) {
        for (int i = 0; i < input.size(); ++i) {
            update(i, input[i]);
        }
    }

    // Get the prefix sum up to index i
    int getSum(int i) {
        i++;  // Convert 0-based index to 1-based index
        int sum = 0;
        while (i > 0) {
            sum += BIT[i];
            i -= (i & -i);  // Move to the ancestor in the binary tree
        }
        return sum;
    }

    // Display the Fenwick Tree
    void displayTree() const {
        for (int i = 1; i < BIT.size(); ++i) {
            cout << BIT[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    vector<int> inputArray = {1, 2, 3, 4, 5};
    int n = inputArray.size();

    FenwickTree fenwickTree(n);

    // Construct Fenwick Tree from the input array
    fenwickTree.constructTree(inputArray);

    // Display the Fenwick Tree array
    cout << "Fenwick Tree Array: ";
    fenwickTree.displayTree();

    return 0;
}
```

4)Segment Tree:-

A segment tree is a binary tree data structure used for efficient querying and updating of elements in an array or a list. It is particularly useful when you need to perform range queries and updates on a dynamic set of data. The main advantage of a segment tree is that it allows these operations to be performed in logarithmic time complexity, making it efficient for a variety of problems.

Here's a brief overview of a segment tree and why it is used:

Structure of a Segment Tree:

- **Tree Representation:** A segment tree is represented as a binary tree where each leaf node corresponds to an element in the input array, and each non-leaf node represents a segment (range) of the array.
- **Leaf Nodes:** The leaf nodes store the actual elements of the array.
- **Non-Leaf Nodes:** Each non-leaf node represents a segment of the array, and it stores information summarizing the values in that segment (e.g., the sum, minimum, maximum).

Purpose and Use Cases:

1. **Range Queries:**
    - **Example Operations:** Find the sum, minimum, maximum, or other aggregate values over a specified range of array elements.
    - **Efficiency:** Segment trees can perform these range queries in logarithmic time, making them efficient for large datasets.
2. **Range Updates:**
    - **Example Operations:** Update the values of elements in a specified range.
    - **Efficiency:** Segment trees efficiently handle range updates by modifying only the affected nodes in the tree.
3. **Dynamic Sets:**
    - **Example Operations:** Insertion and deletion of elements.
    - **Efficiency:** Segment trees provide efficient support for dynamic sets, allowing elements to be added or removed with logarithmic time complexity.
4. **Interval Overlaps:**
    - **Example Operations:** Find all intervals that overlap with a given interval.
    - **Efficiency:** Segment trees are well-suited for problems that involve interval overlaps.

Implementation:

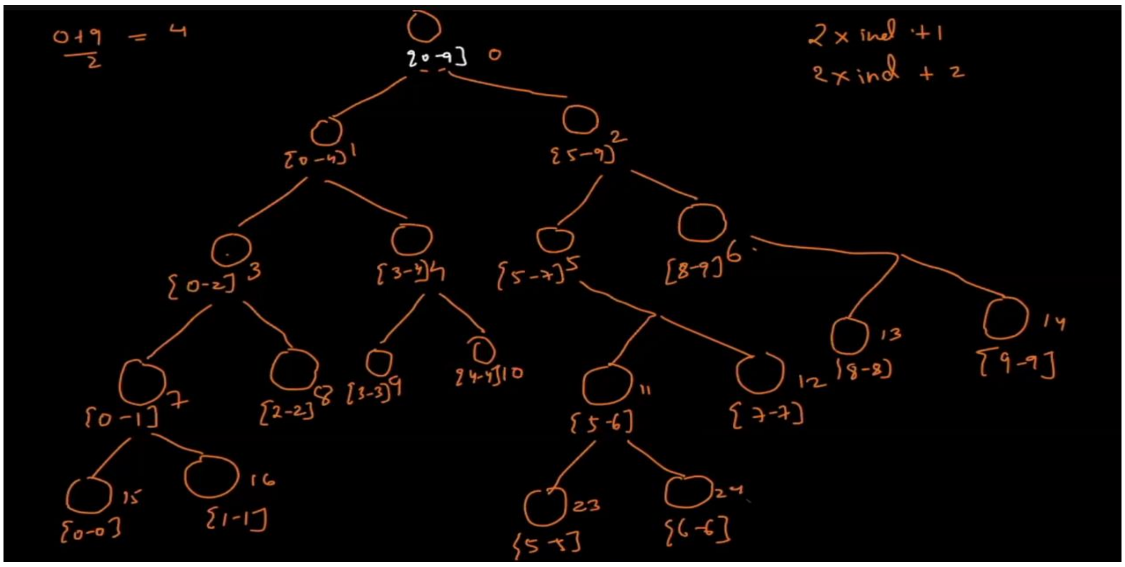
- **Recursive Structure:** The segment tree is often implemented using a recursive approach where each node represents a segment of the array.
- **Lazy Propagation:** In some cases, a technique known as lazy propagation is used to optimize range updates, delaying the actual updates until they are needed.

Example Scenario:

Consider a scenario where you have an array of values, and you want to efficiently compute the sum of elements in a given range. A segment tree can be built for this array, and querying the tree for the sum of a range would be much faster than iterating through the array elements.

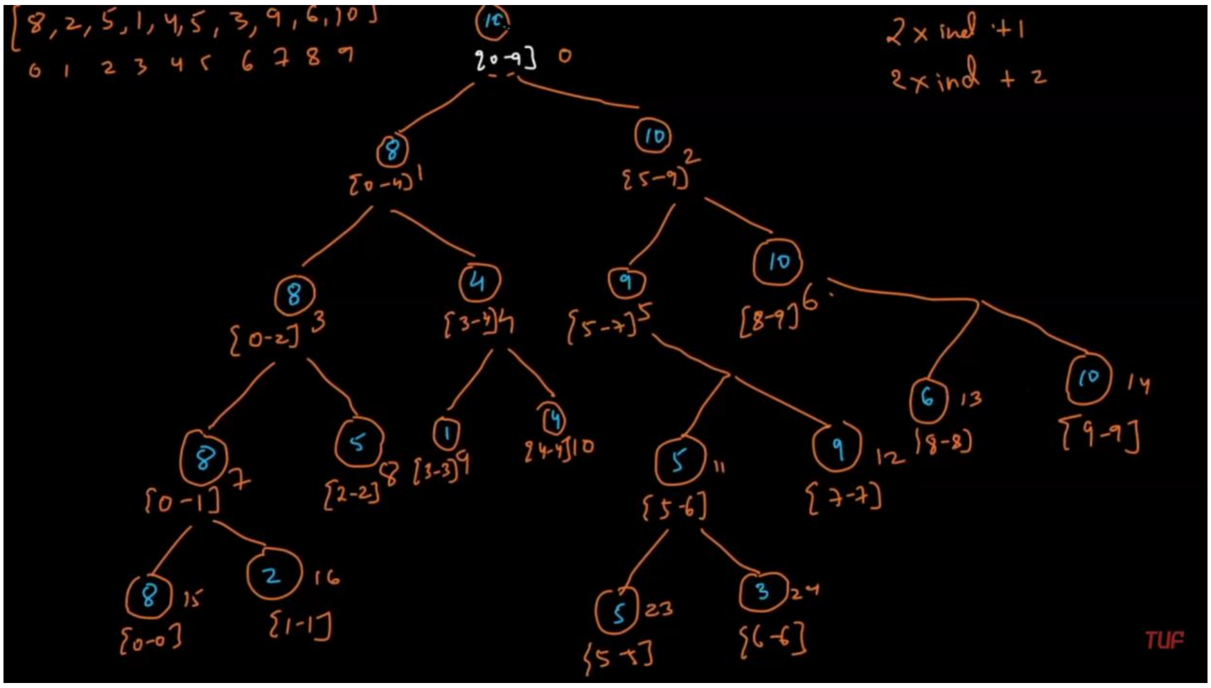
In summary, a segment tree is a versatile data structure that provides efficient solutions to problems involving range queries and updates. Its logarithmic time complexity makes it suitable for a wide range of applications, especially in scenarios where array-like data structures need to be queried or updated efficiently over specified ranges.

Let's say we have to find the sum between the given range of an array. So in the segment tree, we are Creating a tree and in each node, we are storing the sum of elements for a given specific range. Like this:-

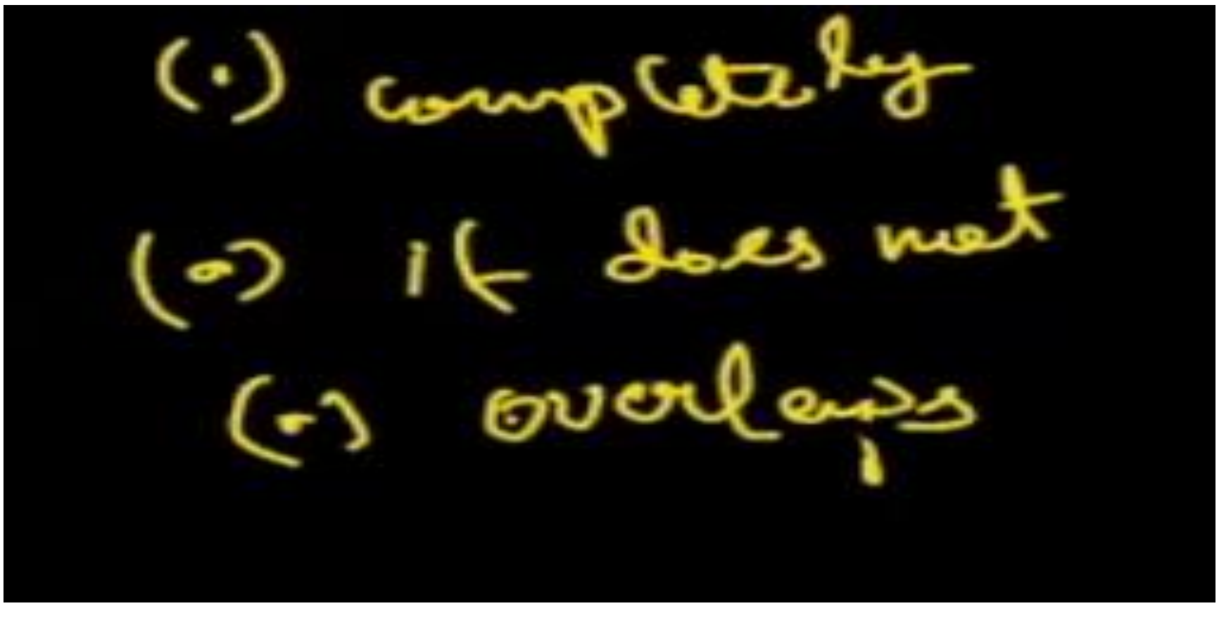


We fill each node according to the question. ex it can store the maximum element of the two nodes, or it can store the sum of nodes, it all depends.

But in this case we have stored the maximum for solving find maximum type of questions:-



Now, when queries are given we have 3 possibilities of that query with provided range of the node and that are:-

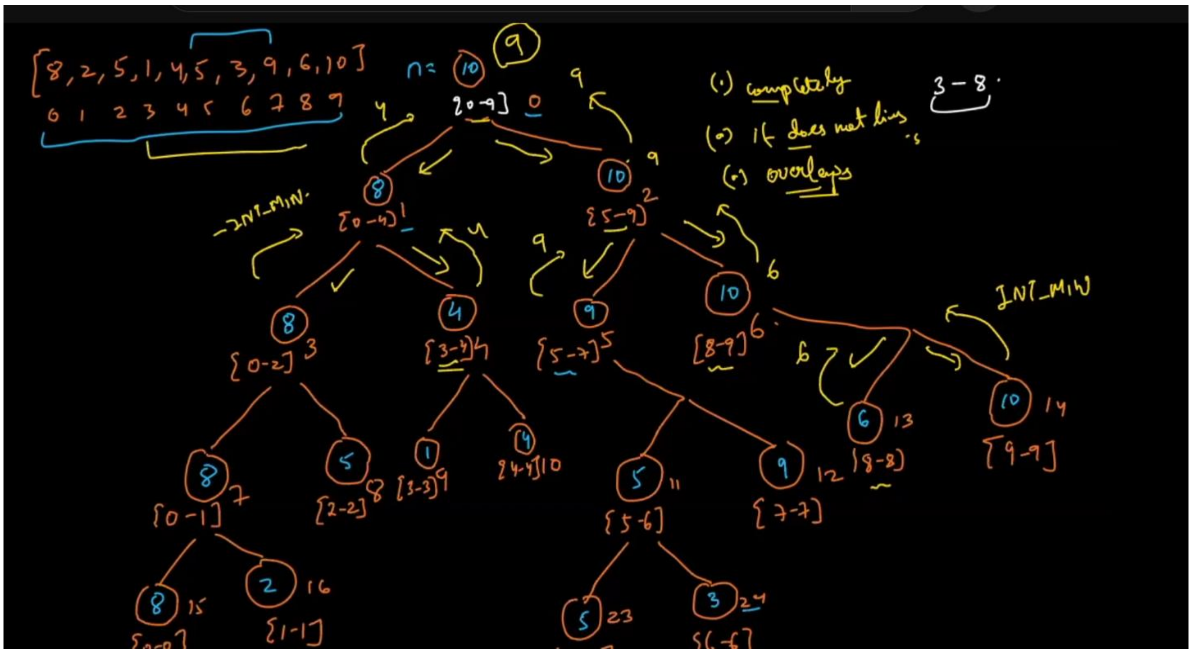

(·) completely
(·) if does not
(·) overlaps

So whenever the given query overlaps with the present node (at which the pointer is initially present) we traverse both sides of the node.

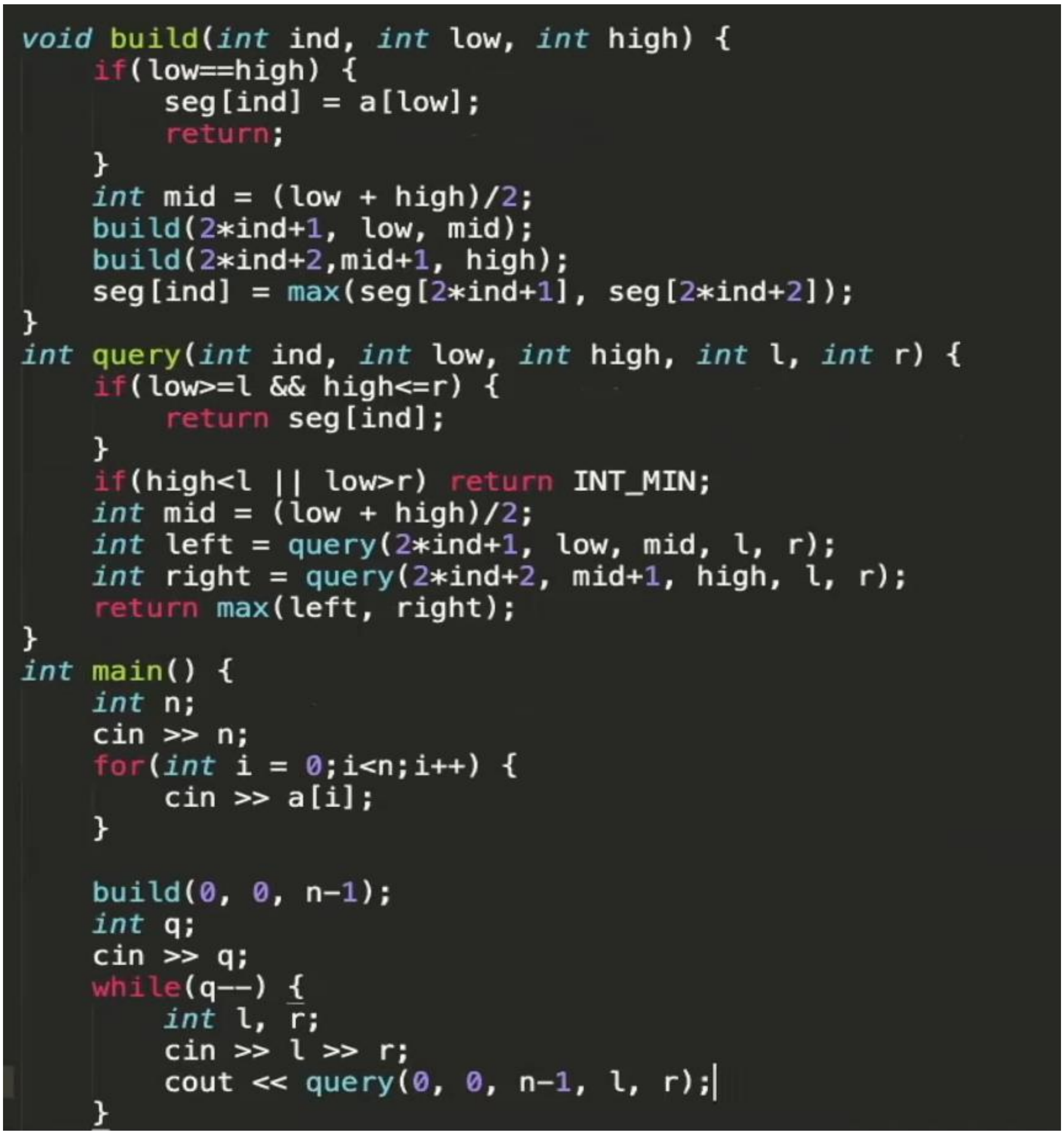
Whenever the present node range does not overlap or completely lies inside the query range, then return to the previous node with or without some value.

Whenever, the present node range does lies completely inside the given query range then return the value of node.

For example the given query in (3 to 8), so the tree will look like this:-



Final code for the finding the maximums:-



```
void build(int ind, int low, int high) {
    if(low==high) {
        seg[ind] = a[low];
        return;
    }
    int mid = (low + high)/2;
    build(2*ind+1, low, mid);
    build(2*ind+2,mid+1, high);
    seg[ind] = max(seg[2*ind+1], seg[2*ind+2]);
}
int query(int ind, int low, int high, int l, int r) {
    if(low>=l && high<=r) {
        return seg[ind];
    }
    if(high<l || low>r) return INT_MIN;
    int mid = (low + high)/2;
    int left = query(2*ind+1, low, mid, l, r);
    int right = query(2*ind+2, mid+1, high, l, r);
    return max(left, right);
}
int main() {
    int n;
    cin >> n;
    for(int i = 0;i<n;i++) {
        cin >> a[i];
    }

    build(0, 0, n-1);
    int q;
    cin >> q;
    while(q--) {
        int l, r;
        cin >> l >> r;
        cout << query(0, 0, n-1, l, r);
    }
```

4)Lazy Propogation:-

Lazy propagation is a technique used in segment trees to optimize updates in scenarios where you need to make multiple updates to a range of elements in the array. Instead of immediately updating all affected nodes in the segment tree, lazy propagation defers the actual updates until they are needed. This can significantly reduce the number of update operations performed during a sequence of range update queries.

Here's a general overview of how lazy propagation works in the context of segment trees:

Basic Concept:

1. **Lazy Propagation Information:**
   - For each node in the segment tree, an additional "lazy" value is maintained.
   - This lazy value represents the pending update that needs to be applied to the entire range represented by the node.
2. **Deferred Updates:**
   - When a range update is requested, the segment tree updates the lazy value of the corresponding nodes without immediately modifying their actual values.
   - The updates are applied lazily when a query operation requires information from a node.
3. **Propagation:**
   - During a query, when a node with a non-zero lazy value is encountered, the update is propagated down to its children, and the lazy value is passed on.
   - This ensures that the pending updates are applied only when necessary.

Example Scenario:

Consider a scenario where you have an array representing heights of buildings, and you want to perform multiple increment operations on a specific range of buildings. Without lazy propagation, you might need to update all affected nodes in the segment tree immediately after each increment operation. With lazy propagation:

1. When an increment operation is requested, the corresponding segment tree nodes are marked with a lazy value representing the increment.
2. The actual updates are deferred until a query operation requires information from the affected nodes.
3. During a query operation, the lazy values are propagated down the tree, and the updates are applied only when necessary.

Benefits:

- **Reduced Update Operations:** Lazy propagation reduces the number of actual updates performed, leading to improved time complexity for a sequence of updates.

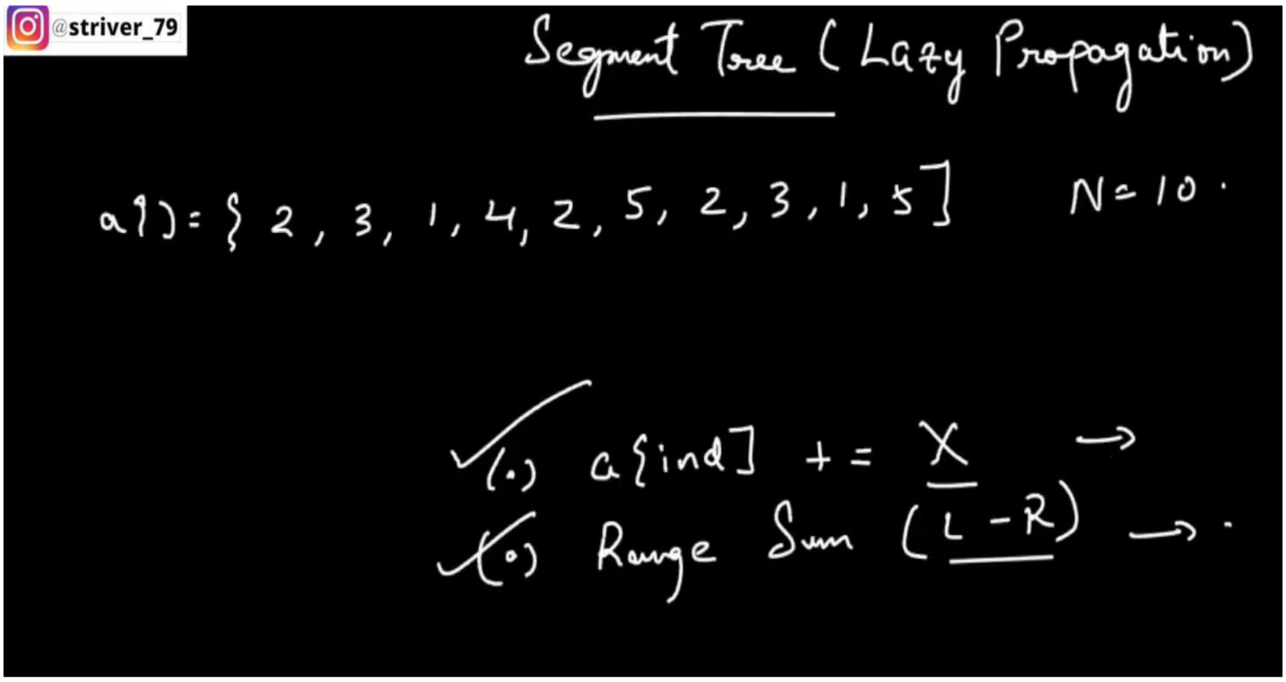
- **Efficient Handling of Range Updates:** It's particularly useful when dealing with range update queries, where a continuous range of elements needs to be modified.

Implementation:

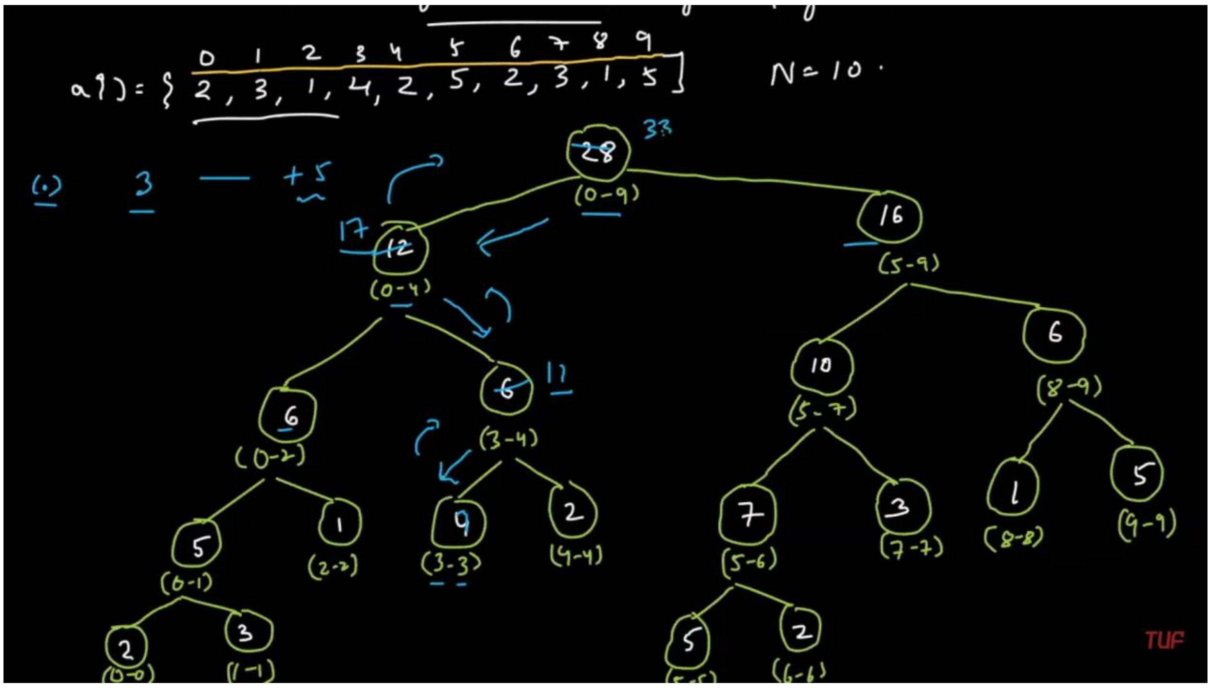
- The implementation of lazy propagation involves augmenting the segment tree nodes with an additional lazy value and updating the propagation logic in the query and update operations.
- Handling lazy propagation often involves recursion, where the lazy values are propagated down the tree during query operations.
- Lazy propagation is commonly used in scenarios where there are multiple update operations in succession, such as range modifications.

Keep in mind that while lazy propagation can significantly optimize certain scenarios, it is not always necessary. Its use depends on the nature of the problem you are solving and the types of queries and updates involved.

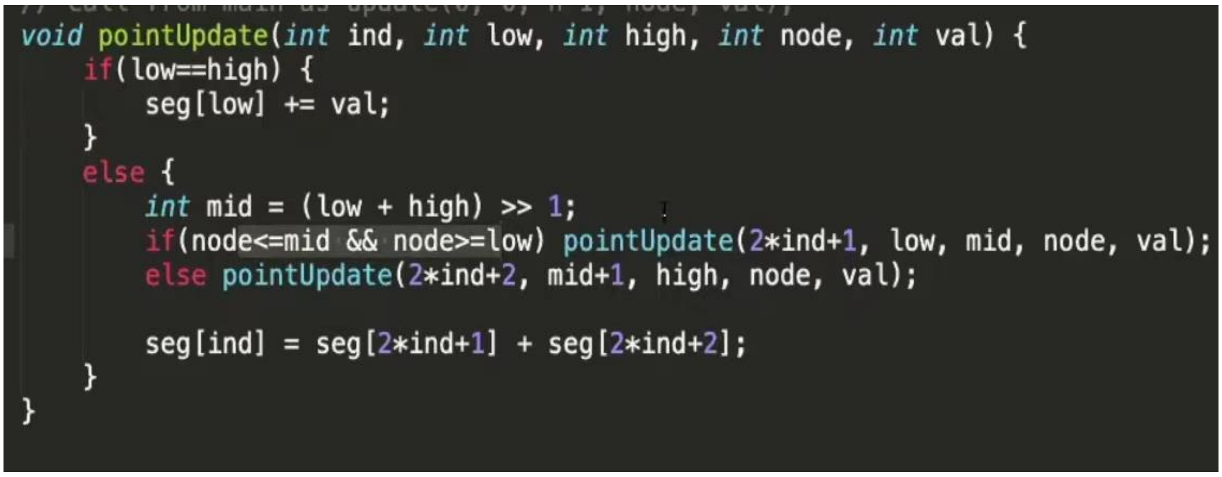
Here we are updating this array:-


@striver_79
Segment Tree (Lazy Propagation)
a[] = {2, 3, 1, 4, 2, 5, 2, 3, 1, 5]
N = 10.
(.) a[ind] += X →
(.) Range Sum (L - R) →.

<u>Point update</u>:-Here, we are updating the value at index 3 by adding 5 to it.



Code:-



```
void pointUpdate(int ind, int low, int high, int node, int val) {
    if(low==high) {
        seg[low] += val;
    }
    else {
        int mid = (low + high) >> 1;
        if(node<=mid && node>=low) pointUpdate(2*ind+1, low, mid, node, val);
        else pointUpdate(2*ind+2, mid+1, high, node, val);

        seg[ind] = seg[2*ind+1] + seg[2*ind+2];
    }
}
```

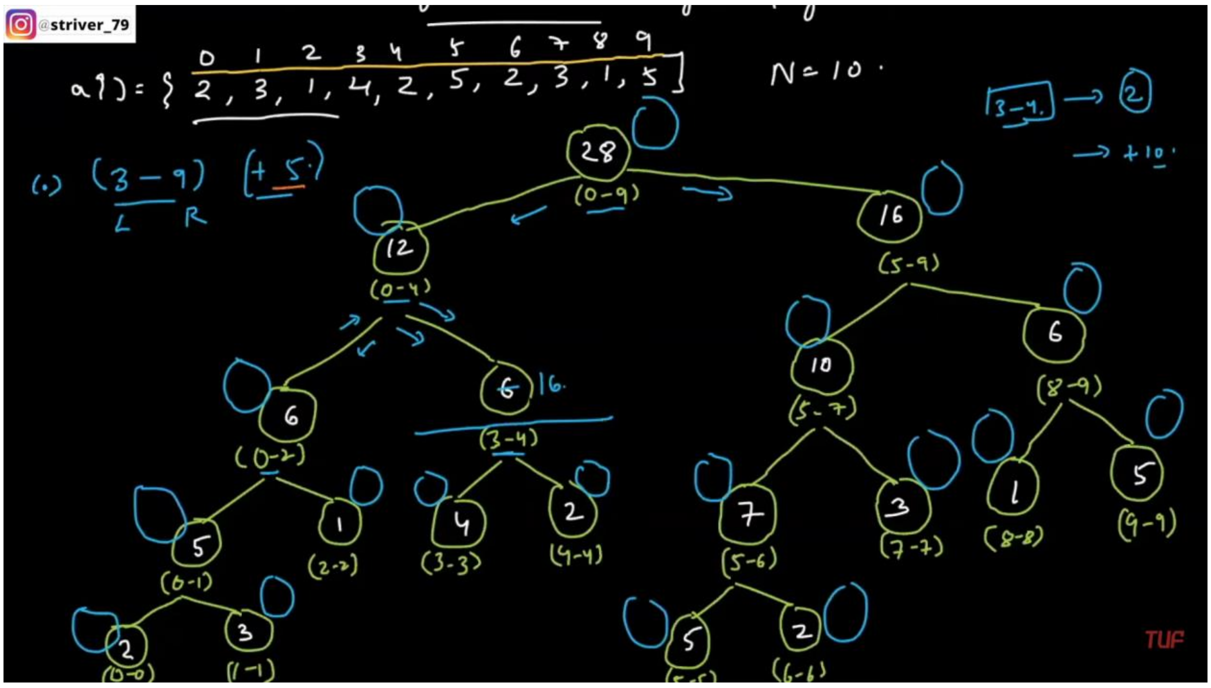
<u>Range Update</u>:-

If the node range lies completely inside the query range then, we simply add/multiply or whatever depends upon the question, two times considering overall change for the child nodes (so we do not have to iterate further down to make a change.

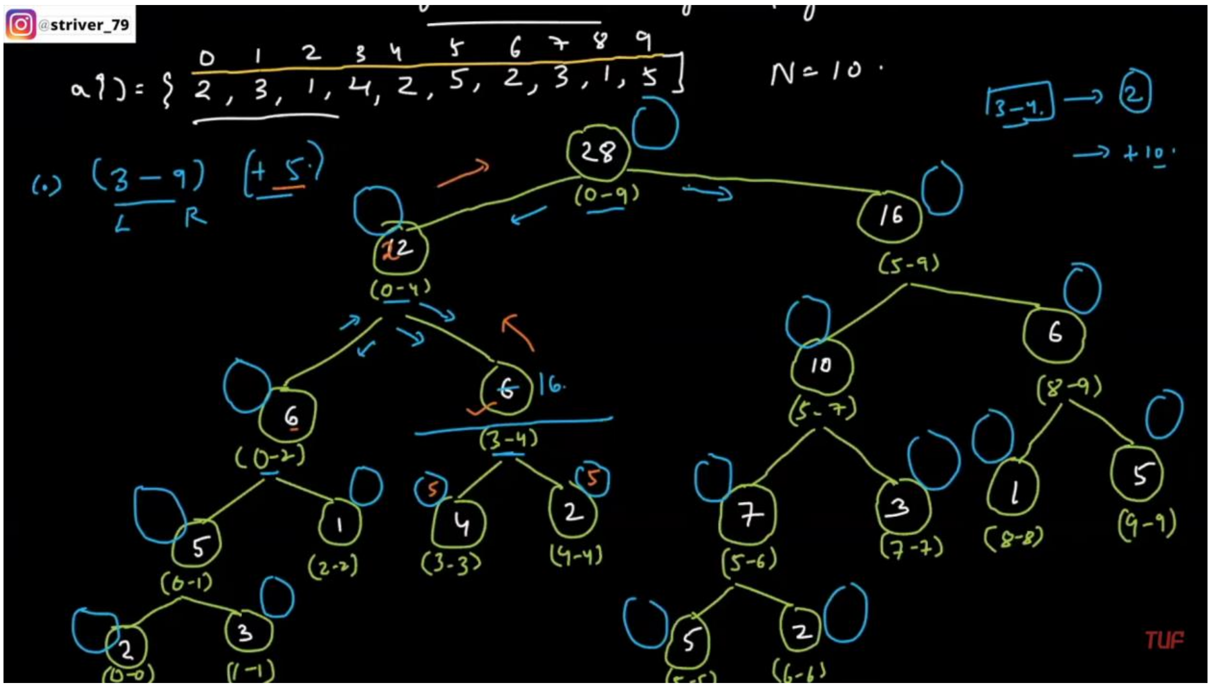
If the query range overlaps the node range, then we iterate both directions(iteration over child nodes)

If the query range lies completely outside the node range simply return back.

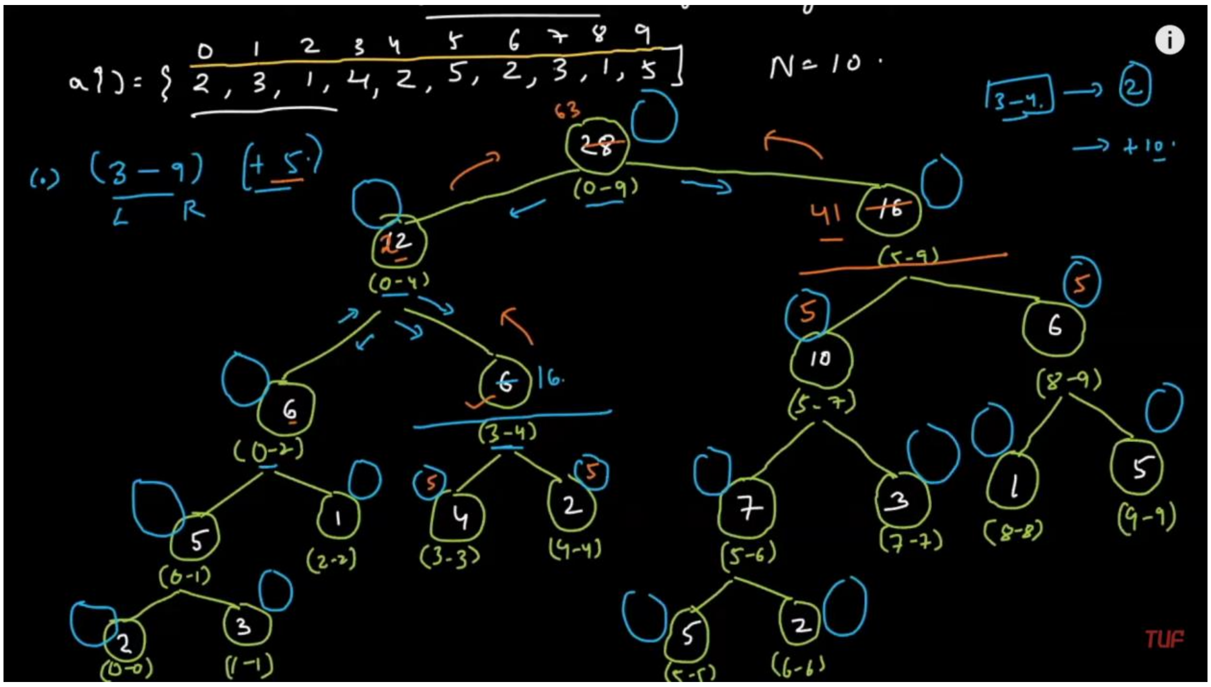
In lazy prorogation we create a replica of a tree and initialized with value 0.



By whatever value, the parent node is updated with, those individual values should be stored in the lazy tree, in this case node of range(3-4) has been updated with value 10 , because 5 is the criteria and it has taken of their children updating by making itself +10 , as the node is storing sum of its child nodes , if both child increases their values by 5 , so the overall change in the parent data is 10 , that's why node increases its value by 10.



First we have updated the tree for (3 to 9) by 5 , and our tree look like this:-



Now we are taking another query for (5 to 8) by adding +3 to the range.

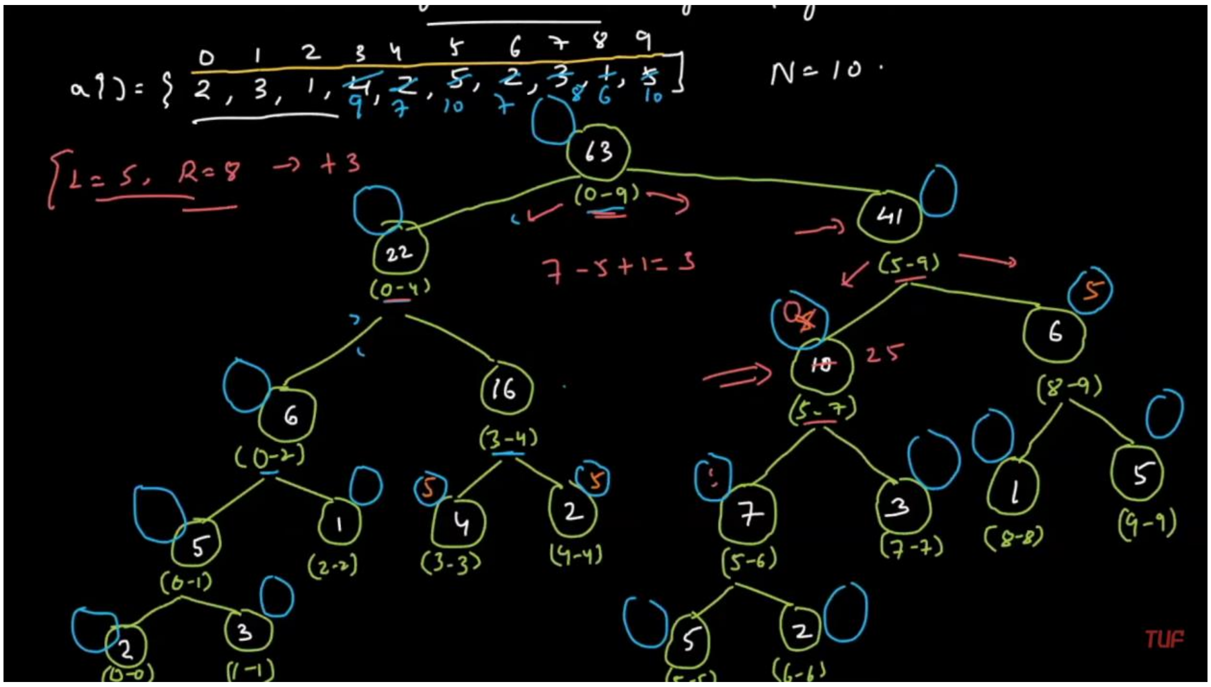
NOTE:-[value added to the parent node will depend upon, how many children it contains]

Now, performing another query, we start iterating and also looking for the values present inside the lazy tree node, whatever the value it contains

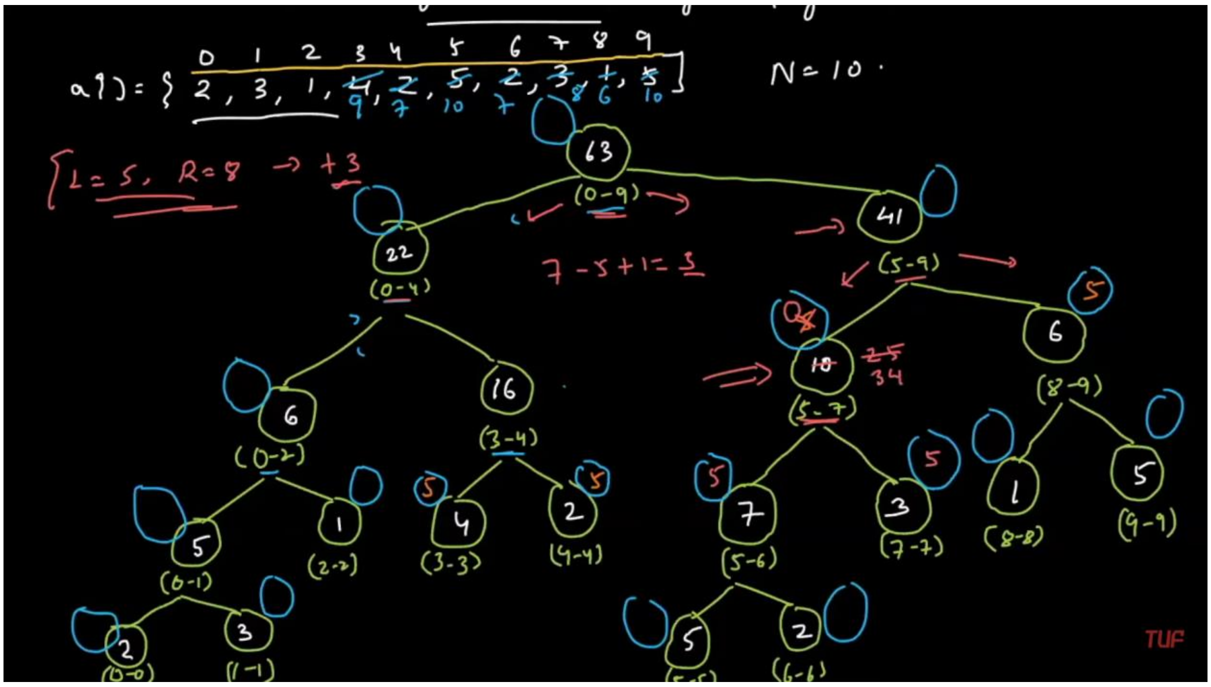
The value of that node will be increased by ((higher limit the node – the lower limit of the node +1)*

(value present in the lazy node)).

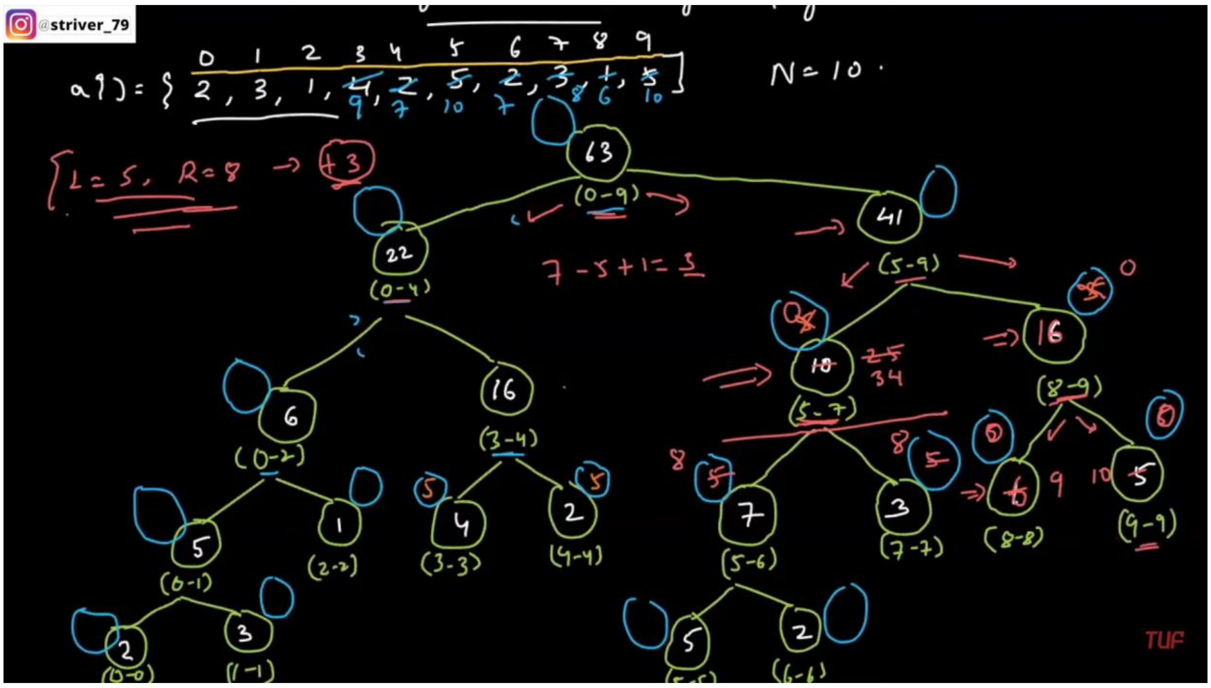
And pass the value of Lazy node to its two child nodes.



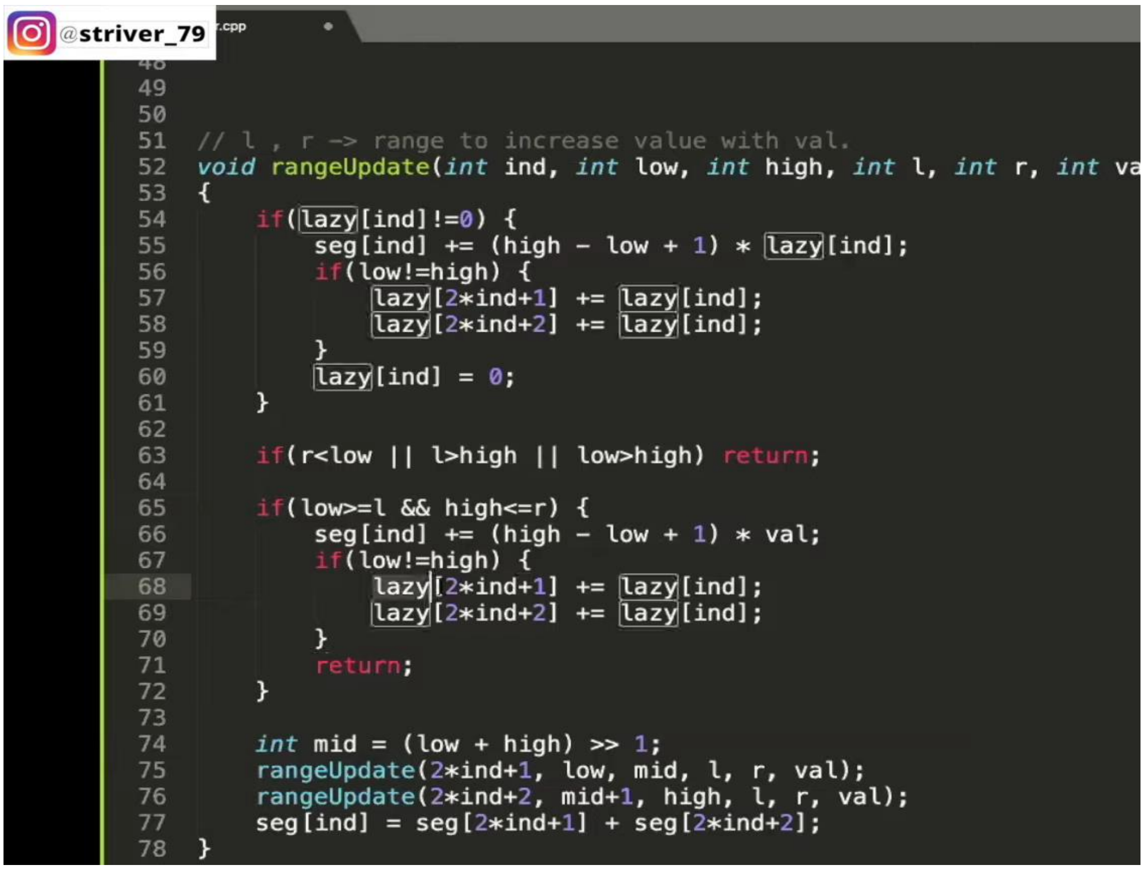
As we see the node of range (5 to 7) already has a lazy node value of 5 so first it is updated then after and assigned the value the lazy node to 0.



As we know the next query is to add +3 so now we have performed that, and added +9 to before value 25 and finally we get 34 as we have added lazy nodes values too , se we pass on this laze node value too , to its child nodes.

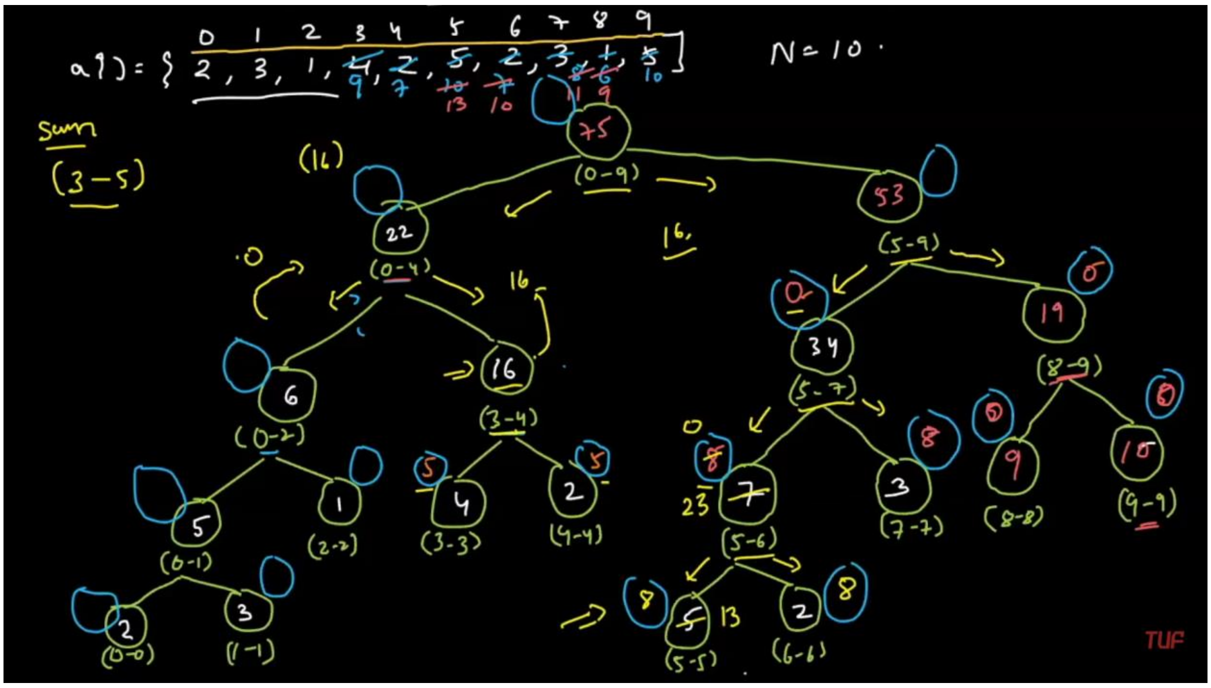


CODE:-



```cpp
// l , r -> range to increase value with val.
void rangeUpdate(int ind, int low, int high, int l, int r, int va
{
    if(lazy[ind]!=0) {
        seg[ind] += (high - low + 1) * lazy[ind];
        if(low!=high) {
            lazy[2*ind+1] += lazy[ind];
            lazy[2*ind+2] += lazy[ind];
        }
        lazy[ind] = 0;
    }

    if(r<low || l>high || low>high) return;

    if(low>=l && high<=r) {
        seg[ind] += (high - low + 1) * val;
        if(low!=high) {
            lazy[2*ind+1] += lazy[ind];
            lazy[2*ind+2] += lazy[ind];
        }
        return;
    }

    int mid = (low + high) >> 1;
    rangeUpdate(2*ind+1, low, mid, l, r, val);
    rangeUpdate(2*ind+2, mid+1, high, l, r, val);
    seg[ind] = seg[2*ind+1] + seg[2*ind+2];
}
```

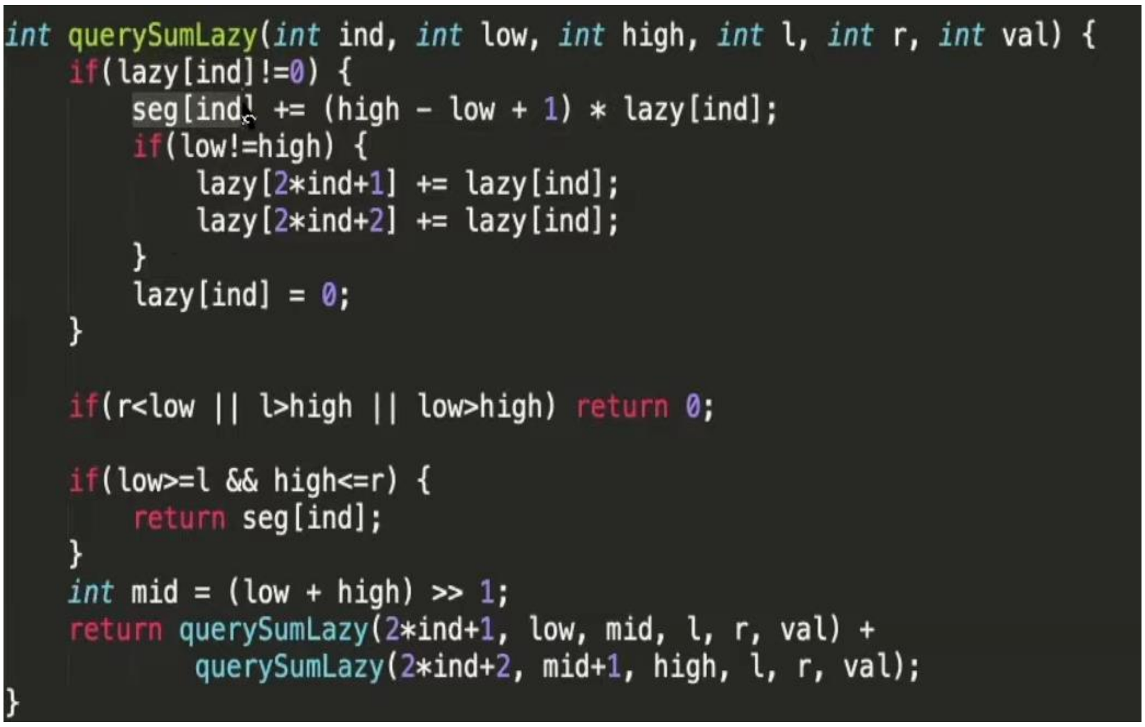
Now, if query asks for the summation of a given range:-



For this example the summation asked for the range (3 to 5), we start iterating , if the node range lies completely inside the query range , then we simply add the node value to the product of its lazy node value and the no. of element that range of a node contains.

Now if the node range overlaps then we iterate to both side, and also update node value .

CODE:-



```
int querySumLazy(int ind, int low, int high, int l, int r, int val) {
    if(lazy[ind]!=0) {
        seg[ind] += (high - low + 1) * lazy[ind];
        if(low!=high) {
            lazy[2*ind+1] += lazy[ind];
            lazy[2*ind+2] += lazy[ind];
        }
        lazy[ind] = 0;
    }

    if(r<low || l>high || low>high) return 0;

    if(low>=l && high<=r) {
        return seg[ind];
    }
    int mid = (low + high) >> 1;
    return querySumLazy(2*ind+1, low, mid, l, r, val) +
            querySumLazy(2*ind+2, mid+1, high, l, r, val);
}
```