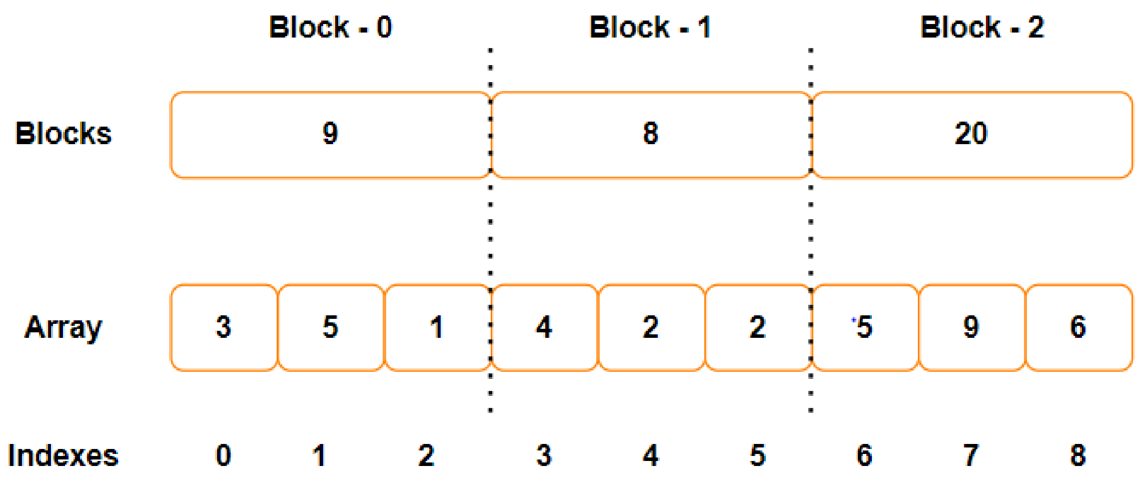
Square Root Decomposition

As stated earlier the idea behind square root decomposition is to decompose or split a structure into smaller chunks, so given an array of size **N**, we decompose this array into smaller **chunks(blocks)** of size **sqrt(N)**.

Since we are dividing our array into smaller chunks/subarrays/blocks of approximately sqrt(N) size(exactly sqrt(N) blocks if N is a perfect square), hence the number of such smaller chunks/blocks will be **N/sqrt(N)** i.e **sqrt(N) blocks**. If N is not a perfect square, then we can take **ceil(sqrt(N))** blocks which will contain some extra elements. However, we can treat them as “dummy elements” of the array whose value does not affect the answer.

After the division of the array let us do some preprocessing i.e for each of the sqrt(N) blocks, let us treat each such block as an individual array whose value is equal to the sum of the elements of the original array present in this block. So if we represent these blocks as an array block[sqrt(N)] of size sqrt(N) then **block[i] = Sum of elements of the original array represented by an ith block.**

For Example:
In the below diagram we have an array of size 10, the size of each block and the number of such blocks will be ceil(sqrt(9)) = 3, with each block representing an array of size 3 and containing the sum of the elements of the array representing that block.



Given an index idx(0 <= idx <= N-1), what will be the index of the block array containing the element represented by Arr[idx] in the original array?
The index of the block array will be **floor(idx / sqrt(N))**.

Given a block index bdx of the block array, what indices of the array will be represented by block[bdx]?
The array Arr from index **bdx*(sqrt(N)) to (bdx+1)*(sqrt(N)) - 1** (both values inclusive).

Hence we can preprocess i.e compute the values of each block by adding the value of the elements of the array belonging to that block -> adding Arr[idx] to block[idx / sqrt(N)].
Since we have preprocessed the answers to each of the blocks(sum of the elements), where the total time taken for preprocessing is **O(N)**, let us see how we can answer our queries of Type 2 efficiently.

The index of the block array representing the indexes L and R of the original array will be **floor(L / sqrt(N))** and **floor(R / sqrt(N))**, let's see how to calculate the sum of the elements from L to R using this block array:

1. If L and R belong to the **same block,** then we can simply traverse the original array from L to R and compute the sum from L to R which will take **sqrt(N)** iterations in the worst case as a block contains sqrt(N) elements.

   For Example: In the above diagram let us say we need to find the sum of elements from index L = 1 to R = 2 since L and R belong to the same block we can traverse the original array from L to R to find sum which equals 6.

2. If L and R belong to **different blocks,** where L is at the **start** of the **block[floor(L / sqrt(N))]** and R is at the **end** of the **block[floor(R / sqrt(N))],** then we can simply iterate from (L / sqrt(N)) to (R / sqrt(N)) in the block array and find the sum of elements i.e we will be making jumps of sqrt(N) and take advantage of preprocessing the compute the overall sum from L to R which will take **sqrt(N)** iterations in the worst case as we only have sqrt(N) blocks.

   For Example: In the above diagram, let L = 0 and R = 5, then we can use the preprocessed values and calculate the index of the left block = floor(0 / 3) = 0 and index of the right block = floor(5/3) = 1 and find the sum of block array from 0 to 1(left block to right block) = 17

3. If L and R belong to **different blocks** with L and R present at **some position**(not necessarily at the start and end of the block) within the blocks represented by block[floor(L / sqrt(N))] and block[floor(R / sqrt(N))], then we can first compute the sum of elements present in block[floor(L / sqrt(N))] i.e from **L to (floor(L / sqrt(N)+1))*(sqrt(N)) - 1** as L is not necessarily at the beginning of this block and then we can make jumps of sqrt(N) through blocks till **floor(R/sqrt(N) - 1)** i.e the block previous to the block represented by R and then again find the sum of elements present in block[floor(R / sqrt(N))] i.e from **(floor(R/sqrt(N)))*(sqrt(N)) to R**.
   This will take (sqrt(N) + sqrt(N) + sqrt(N)) iterations in the worst case as iterating over the block represented by L and the block represented by R individually takes **O(sqrt(N))** iterations each and there can be at most sqrt(N) jumps in between the block indices.

   For Example: Let L = 1 and R = 7, the index of the left and right blocks will be floor(1 / 3) = 0 and floor(7 / 3) = 2 since L and R are at the partial ends of the block so we can first

find the sum of elements from 1 to floor(1 / 3 + 1)*3 -1 = 2 (the end of the block in which L is present) = 6, then we can use the preprocessed value of the block to find sum from 3 to 5 = 8 and again find the sum of elements from floor(7 / 3)*3 = 6(the start of the block in which R is present) to 7 = 14.
So sum = 6 + 8 + 14 = 28

Now we have an efficient algorithm to answer queries of **Type 2** in **O(sqrt(N))** time complexity in the worst case, but how to perform queries of Type 1 on this block array?
Given an index **idx** and value **val,** we can simply update the block representing this index as:

1. Removing the contribution of the old value Arr[idx] from the block
   block[floor(idx / sqrt(N))] = block[floor(idx / sqrt(N))] - Arr[idx]
2. Adding the new value to the block
   block[floor(idx / sqrt(N))] = block[floor(idx / sqrt(N))] + val
3. Finally updating the given array to the new value.
   Arr[idx] = val

Hence, queries of **Type 1** can be performed in **O(1)** time.

Pseudocode:

```
/*
        The function takes input array 'Arr', the size of the array 'N', the block array
        'block' and the block size 'block_sz' and finds the preprocessed values to store in
        the block array.
*/
function preprocess(Arr, N, block, block_sz)

        /*
                Iterating over the elements of the array and adding the element Arr[idx]
                to the block containing it i.e block[idx / block_sz].
        */
        for idx = 0 to N - 1
                block[idx / block_sz] = block[idx / block_sz] + Arr[idx]

/*
        The function takes input array 'Arr', range indices 'L' and 'R', block array 'block'
        and block size 'block_sz' and returns the sum from L to R.
*/
function getSum(Arr, L, R, block, block_sz)

        //   Initializing sum to 0
        sum = 0

        //   Finding the indices of blocks containing L and R as 'leftBlock' and 'rightBlock'
```

```
    leftBlock = L / block_sz
    rightBlock = R / block_sz

    //   If the range [L, R] falls in the same block then simply iterate from L to R
    if leftBlock equals rightBlock
        for idx = L to R
            sum = sum + Arr[idx]

    //   If the range [L, R] falls in different blocks
    else
        //   Finding the sum of elements present in the block ‘block[leftBlock]’
        for idx = L to (leftBlock+1)*block_sz - 1
            sum = sum + Arr[idx]

        /*
            Finding the sum of elements through the blocks ‘leftBlock + 1’ to
            ‘rightBlock - 1’ taking advantage of the preprocessing.
        */
        for idx = leftBlock + 1 to rightBlock - 1
            sum = sum + block[idx]

        //   Finding the sum of elements present in the block ‘block[rightBlock]’
        for idx = rightBlock*block_sz to R
            sum = sum + Arr[idx]

    return sum

/*
    The function takes input array ‘Arr’, index ‘idx’, value ‘val’, block array ‘block’ and
    block size ‘block_sz’ and updates the value of block and array at given index.
*/
function update(Arr, idx, val, block, block_sz)

    block[idx / block_sz] = block[idx / block_sz] - Arr[idx]
    block[idx / block_sz] = block[idx / block_sz] + val
    Arr[idx] = val



/*
    The function takes input array ‘Arr’, the size of the array ‘N’, and number of
    queries ‘Q’
*/
function solve(Arr, N, Q)
```

```
/*
        Initializing block size denoting the size of each block and the total
        number of blocks
*/
block_sz = ceil(sqrt(N))

//  Declaring a block array initialized to 0
block[block_sz] = {0}

//  Preprocessing the values for the block array
preprocess(Arr, N, block, block_sz)

//  Iterating over the number of queries Q
for idx = 1 to Q
        //  Taking input of the “type” of query
        input(type)

        //  Taking input index idx and value val
        if type equals 1
                input(idx,val)
                update(Arr, idx, val, block, block_sz)

        //  Taking input left and right indices L and R
        else
                input(L, R)
                sum = getSum(Arr, L, R)
                print(sum)
```

Time Complexity: O(N + Q*(sqrt(N))), where it takes O(N) time for preprocessing and each query can take O(sqrt(N)) time in the worst case.

Let us try to solve another variant of this problem which again has two types of queries on an array **Arr** of size **N**:
Type 1: Add a value **val** to all the elements in the range [L, R] (0 <= L <= R <= N - 1)
Type 2: Find the value of the element at index **idx**.

We can again use sqrt decomposition to handle queries of Type 1 efficiently. We will decompose the array into sqrt(N) blocks with each block containing sqrt(N) elements of the original array and initialize them to 0. For every query of **Type 1**, we add the given value **val** to all the blocks which contain the elements of the array that lie completely inside the range [L, R] and add **val** to the original array Arr[i] for the tails of the intervals(Case 3).

For every query of **Type 2**, the answer will be Arr[idx] + Block[idx / sqrt(N)].