MO’s Algorithm (Query Square Root Decomposition) | Set 1 (Introduction)

```
// Program to compute sum of ranges for different range
// queries
#include <bits/stdc++.h>
using namespace std;

// Variable to represent block size. This is made global
// so compare() of sort can use it.
int block;

// Structure to represent a query range
struct Query
{
    int L, R;
};

// Function used to sort all queries so that all queries
// of the same block are arranged together and within a block,
// queries are sorted in increasing order of R values.
bool compare(Query x, Query y)
{
    // Different blocks, sort by block.
    if (x.L/block != y.L/block)
        return x.L/block < y.L/block;

    // Same block, sort by R value
    return x.R < y.R;
}

// Prints sum of all query ranges. m is number of queries
// n is size of array a[].
void queryResults(int a[], int n, Query q[], int m)
{
    // Find block size
    block = (int)sqrt(n);

    // Sort all queries so that queries of same blocks
    // are arranged together.
    sort(q, q + m, compare);

    // Initialize current L, current R and current sum
    int currL = 0, currR = 0;
    int currSum = 0;

    // Traverse through all queries
    for (int i=0; i<m; i++)
    {
        // L and R values of current range
        int L = q[i].L, R = q[i].R;

        // Remove extra elements of previous range. For
        // example if previous range is [0, 3] and current
        // range is [2, 5], then a[0] and a[1] are subtracted
        while (currL < L)
        {
            currSum -= a[currL];
            currL++;
```

```
        }

        // Add Elements of current Range
        while (currL > L)
        {
            currSum += a[currL-1];
            currL--;
        }
        while (currR <= R)
        {
            currSum += a[currR];
            currR++;
        }

        // Remove elements of previous range.  For example
        // when previous range is [0, 10] and current range
        // is [3, 8], then a[9] and a[10] are subtracted
        while (currR > R+1)
        {
            currSum -= a[currR-1];
            currR--;
        }

        // Print sum of current range
        cout << "Sum of [" << L << ", " << R
             << "] is "   << currSum << endl;
    }
}

// Driver program
int main()
{
    int a[] = {1, 1, 2, 1, 3, 4, 5, 2, 8};
    int n = sizeof(a)/sizeof(a[0]);
    Query q[] = {{0, 4}, {1, 3}, {2, 4}};
    int m = sizeof(q)/sizeof(q[0]);
    queryResults(a, n, q, m);
    return 0;
}
```

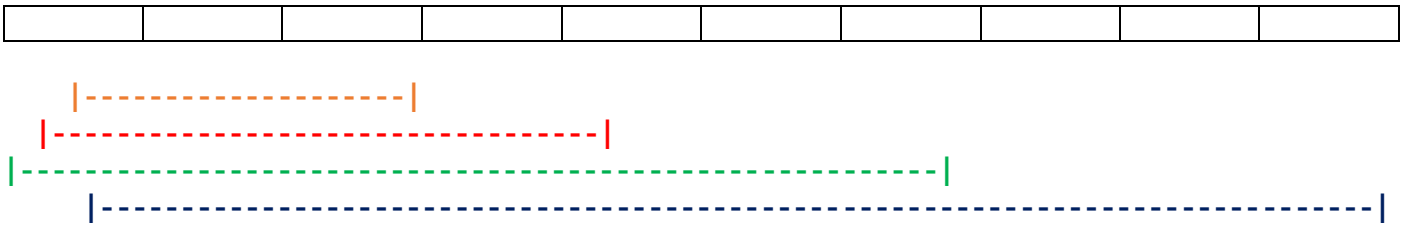
Q1. Why is the block size of sqrt(N) required ?

```
bool compare(Query x, Query y)
{
    // Different blocks, sort by block.
    if (x.L!=y.L)
        return x.L < y.L;

    // Same block, sort by R value
    return x.R < y.R;
}
```

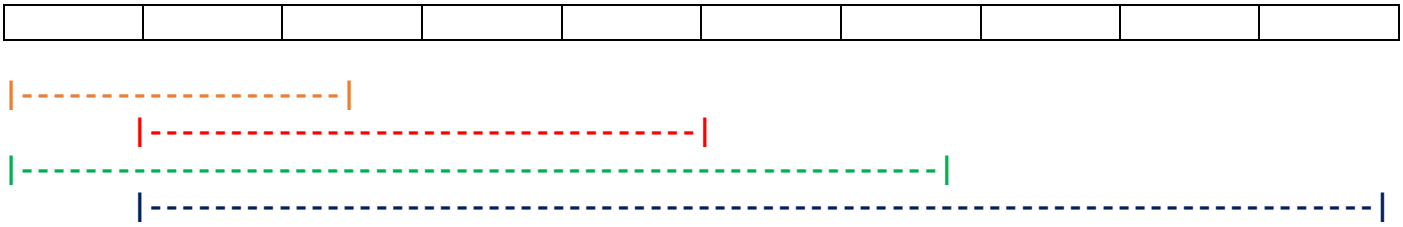
Let's suppose we have the compare function like above. It will be Q*N where Q is the number of query and N is the size of the array [for certain cases].

Q2. Why does the right pointer(R) move O(N*sqrt(N)) times at worst?



For each block the (R) pointer will move till N at max.
For sqrt(N) blocks it will move N*sqrt(N) times.

Q3. Why does the left pointer(L) move O(Q*sqrt(N)) times at worst?



As can be seen from the image the (L) pointer will move sqrt(N) times back & forth in each query and this will happen Q times(Q queries).