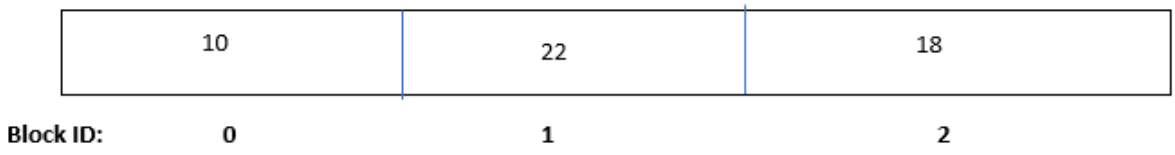
Range Queries III

Square root decomposition

In this technique, we divide the array into blocks of √n elements. When we process a query from l to r, the idea is to store the pre-processed answer in the blocks array so when we get a block in a range of l to r we do not iterate through all the √n elements of the block rather we get the value from block array in O(1) time. This method reduces the time complexity by √n . Let us see this with an example of a Range Sum Query.

There are 10 elements in the array hence n = 10, now we divide this array into blocks of 3 elements each and 4 elements in the last block.

Our block array will look like this:



Preprocessing is the basic concept of sqrt decomposition. We'll partition the array ‘a’ into blocks of roughly length sqrt(N) and pre-calculate the sum of the elements in each

of the blocks. Let's say for a block, ‘k’ sum is b[k]. We will find the value of b[k] using the formula given below.

$$b[k] = \sum_{i=k \cdot s}^{\min(n-1,(k+1) \cdot s-1)} a[i]$$

As a result, we've figured out the values of b[k](this will cost us O(N) operations). But how can they assist us in answering the range queries [l,r]?

If the interval [l, r] is long enough, it will contain numerous full blocks, and we may find the sum of their elements in a single operation for those blocks. As a result, the interval will only contain parts of two blocks, and the sum of elements in these parts will be straightforward to calculate.

We only need sum of the two tail elements that are

$[l \dots (k+1) \cdot s - 1]$ and $[p \cdot s \dots r]$

We can use the formula below to solve the problem

$$\sum\_i = l^r a[i] = \sum\_i = l^{(k+1) \cdot s-1} a[i] + \sum\_i = k + 1^{p-1} b[i] + \sum\_i = p \cdot s^r a[i]$$

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

int main()
{
    /*Input */
    int n;
    cin >> n;
    vector<int> a(n, 0);
    for (int i = 0; i < n; i++)
```

```
{
  int x;
     cin >> x;
  a[i] = x;
}

/*Precalculations.*/
int sqrtValue = (int) sqrt(n + .0);

/*This is the size of one block */
int len = sqrtValue + 1;
vector<int> b;
b.resize(len);

/*Finding sums */
for (int i = 0; i < n; ++i)
  b[i / len] += a[i];

int q;
cin >> q;

/*Queries*/
for (int i = 0; i < q; i++)
{
  int l, r;



     cin >> l >> r;

  int sum = 0;
  for (int i = l; i <= r;)
       if (i + len - 1 <= r && i % len == 0)

         {   /*If the complete block 'i' belongs to[l,r]*/
             sum = sum + b[i / len];
             i = i + len;
```

```
            }
      else
         {
             sum += a[i];
             ++i;
         }
         cout << sum << endl;
    }
}
```

Complexities

Time Complexity

O(Q * sqrt(N)), where ‘N’ is the size of the array and ‘Q’ is the number of queries.

Reason: We are first pre-processing the sums that will cost us O(N) time then for every query we will run a loop from ‘L’ to ‘R’ which even in the worst case will cost us O(sqrt(N)) time. Thus, the overall time complexity to

O(Q * sqrt(N)) + O(N) ~ O(Q * sqrt(N)).

Space Complexity

O(N), where ‘N’ is the size of the array.

Reason: This is the space used by sum array ‘b’.

Mo’s Algorithm

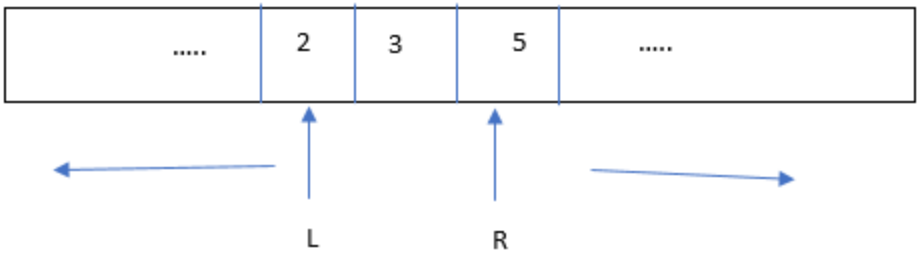
We have seen how we can solve Range queries () using sqrt decomposition. Similar to sqrt decomposition, we have Mo's Algorithm. Now, we must precompute the

answers for each block in a standard sqrt decomposition and merge them when answering queries. In some cases, the merging process can be rather difficult.

For example, if each query requests the mode of its range (the number that appears the most often). This would necessitate each block storing the count of each integer in some sort of data structure, and we can no longer conduct the merge step quickly enough.

Mo's method takes an entirely new approach to answering these types of questions quickly, as it only keeps track of one data structure and only performs simple and quick operations on it. The concept is to respond to questions in a specific order based on the indexes. All queries with the left index in block 0 will be answered first, followed by queries with the left index in block 1, and so on. Also, we'll have to respond to a block's questions in a specific sequence, specifically by sorting the queries by their right index. We will only use one data structure.

The information about the range will be stored in this data structure. This range will be empty at the start. When we want to answer the following inquiry (in the specific order), we simply add or remove elements on both sides of the current range until it becomes the query range. We just have to add or remove a single piece at a time this way, which should be relatively simple operations in our data structure.



But, This is only possible if we are allowed to answer queries in offline mode because we change the order in which they are answered.

Problem Statement

We are given an array and some queries. Our task is to find the sum of every query range.

Approach:

Now the naive approach would be to loop over the given range(L, R) and find a sum for every query. But we are not gonna use that here. We will see how we can use Mo's algorithm to solve this problem. Let's say that 'ARR' is given to us and its size is 'N'.

MO's algorithm works by pre-processing all queries such that the results of one query can be used in the next. The steps are listed below.

- All queries should be sorted in such a way that queries with 'L' values ranging from 0 to sqrt(N) – 1 are grouped together, followed by queries with 'L' values ranging from n to 2 * sqrt(N) – 1, and so on. Within a block, all queries are ordered in increasing order of R values.
- Process each question one at a time, making sure that each one uses the sum computed in the preceding query.
    - Let 'SUM' be the total sum of the previous queries.
    - Remove any remaining entries from the preceding query. If the prior query was [0, 8], and the current query is [3, 9], we deduct ARR[0], ARR[1], and ARR[2] from the 'SUM'.

- To the current query, add new elements. We add a[9] to total in the same example as before.

Let's look at the implementation of the problem.

Code:

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

/*To store the size of a particular block. */
int block;

/*Comparator function to sort queries */
bool compare(vector<int> x, vector<int> y)
{
    /*Sorting by block*/
    if (x[0] / block != y[0] / block)
        return x[0] / block < y[0] / block;

    /*Based on R values. */
    return x[1] < y[1];
}

void queryResults(vector<int> arr, vector<vector< int>> query)
{
    block = (int) sqrt(arr.size());

    /*Sorting queries. */
    sort(query.begin(), query.begin() + query.size(), compare);

    int currL = 0, currR = 0;
    int currSum = 0;
```

```
    /*Traversing through queries */
    for (int i = 0; i < query.size(); i++)
    {
        int L = query[i][0], R = query[i][1];

        /*Removing extra Elements*/
        while (currL < L)
        {
            currSum -= arr[currL];
            currL++;
        }

        /*Adding Elements that are in current Range */
        while (currL > L)
        {
            currSum += arr[currL - 1];
            currL--;
        }

        while (currR <= R)
        {
            currSum += arr[currR];
            currR++;
        }

        /*Removing Elements of previous ranges */
        while (currR > R + 1)
        {
            currSum -= arr[currR - 1];
            currR--;
        }

        /*Printing the sum */
        cout << "Sum of[" << L << ", " << R <<
            "] is " << currSum << endl;
    }
```

```
}

// Driver program
int main()
{
    int n;
    cin >> n;
    vector<int> arr(n, 0);

    /*Input */
    for (int i = 0; i < n; i++)
    {
        int x;
        cin >> x;
        arr[i] = x;
    }

    int q;
    cin >> q;
    vector<vector < int>> queries;
    for (int i = 0; i < q; i++)
    {
        int l, r;
        cin >> l >> r;
        queries.push_back({ l, r });

    }

    queryResults(arr, queries);
    return 0;
}
```

Complexities

Time Complexity

O((N + Q) * sqrt(N)), where ‘N’ is the size of the array and ‘Q’ is the number of queries.

Reason: We can see that the index variable for ‘R’ changes at most O(N * sqrt(N)) times even in the worst-case and similarly for ‘L’ it changes its value at most O(Q * sqrt(N)) times Thus, the overall time complexity to O(N * sqrt(N)) + O(Q * sqrt(N)) ~ O((N + Q) * sqrt(N)).

Space Complexity- O(1)

Reason: We are not using any external space.