P THINK FAST S

Competitive Programming

From Problem 2 Solution in O(1)

Query Square Root Decomposition Algorithms

MO’s Algorithm (and other one)

Mostafa Saad Ibrahim

PhD Student @ Simon Fraser University

Range Query Problems

- Given array of N Numbers and Q queries [Start-end], find in the range/interval:
  - range sum/max/min/average/median/lcm/gcd/xor
  - number of elements repeated K times (k = 1 = distinct)
  - **position** of 1st index with **accumulation >=** C
  - the **smallest** number < S (or their count)
  - Value repeats **exactly once** (use xor) or **most frequent**
  - Find the kth elemnt in the sorted distinct list of range
- Brute force is O(NQ), can we do better?
  - Preprocessing algorithms / Data Structures

Range Query Problems

- Data Structures & Algorithms
  - DS: BIT, Segment Tree, heaps, BBST
  - Algos: Square root decomposition, ad hoc preprocessing
- Applying data structures / algorithms
  - Some of them will be the **easiest**
  - While others will be **harder to apply**
  - And some of them might be **impossible** to use
  - Some of them can be easy to apply, but not efficient
  - Square root decomposition: Typically easy to apply (if doable), but for small N (e.g. < 10000-40000)
  - So **think** about possible **choices** before going with one

Other Concerns

- Static vs Dynamic arrays (update operation)
  - Sometimes we are given a static array
  - Then queries are just given ranges to compute F
  - Sometimes, you have **update operations**
  - Update position 10 with value 157
  - Some algorithms that work for static case only
- Online vs offline processing
  - Sometimes you can read all queries and sort them
  - Then answering might be more efficient
  - This is always doable in competitions (read input file)
  - Sometimes **update operation** makes that impractical

MO’s Algorithm

- A sqrt decomposition algorithm
  - Offline processing algorithm
  - All what it does is just **sorting** the queries
  - Then when u process them, you make use of the **overlapping queries** to avoid precomputing
  - **When to use**: Assume you know the answer for interval [12-30], can you get the answer for [14, 37]?
  - Yes, remove positions 12, 13 and add positions 31-37
  - Think in query that compute sum? It is trivial to add/remove values corresponding to these positions
  - For range min (rmq)? one may use multiset to add/remove

MO’s Algorithm

- We will follow this nice explantation style
- Problem: For a given range, find # of elements repeated **at least** 3 times
  - Let A = {1, 2, 3, 1, 1, 2, 1, 2, 3, 1}
  - (L=0, R=4) = [1, 2, 3, 1, 1] = 1
  - (L=1, R=8) = [2, 3, 1, 1, 2, 1, 2, 3] = 2 (for 1, 2 values)
- For simplicity, assume # of queries = N too

Brute force - v1

- For a query, loop on its range, count frequencies and see ones >= 3. O(N^2)

```
for each query:
  answer = 0
  count[] = 0
  for i in {l..r}:
    count[array[i]]++
    if count[array[i]] == 3:
      answer++
```

Brute force - v2

- Let's do a slight modification
  - When computing a query, make use of the previous one
  - Assume you know the answer for range [12-30].
  - Let's compute from it the answer for [14, 37]?
  - remove positions 12, 13 and add positions 31-37
  - So let's define add and remove functions

```
add(position):
  count[array[position]]++
  if count[array[position]] == 3:
    answer++
```

```
remove(position):
  count[array[position]]--
  if count[array[position]] == 2:
    answer--
```

Brute force - v2

```
currentL = currentR = answer = 0
count[] = 0

for each query:
  while currentL < L:
    remove(currentL), currentL++

  while currentL > L:
    add(currentL), currentL--

  while currentR < R:
    add(currentR), currentR++

  while currentR > R:
    remove(currentR), currentR--

output answer
```

- Assume previous query (L=3, R=10)
- then currentL=3 and currentR=10

- New query: (L=5, R=7)
- Then we need to remove: 3, 4
- And remove: 10, 9, 8

- Then
- For currentL, we either add/remove
- For currentR, we either add/remove
- So 4 loops to cover all cases
- For correctness, order should be:
- **remove, add, add, remove**
- See code

MO’s Algorithm

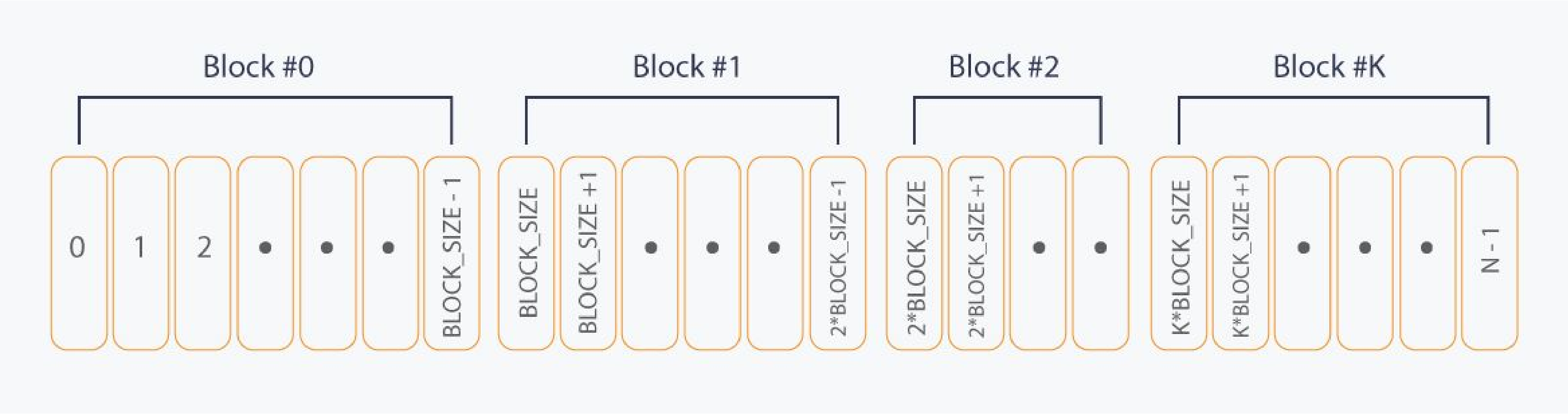
- MO’s algorithm just tells you to
  - Read all queries first
  - Reorder in a way that maximize our overlapping parts from a query to another
  - For example if queries are: (2, 10), (50, 70), (3, 12)
  - If you proceed that order you iterate on all ranges
  - But if ordered to (2, 10), (3, 12), (50, 70)
  - The 2nd range (3, 12) make use of **overlap** with (2, 10)
  - MO’s described a simple ordering that guarantees O((N+M)SQRT(N)) complexity

MO's Algorithm

- Ordering
  - Divide the array to blocks each of length sqrt(N)
  - Each query is assigned to 1 block
  - It is the block where its **Query Left** exists
  - Query [L, R] has a block idx = query.L / sqrt(N)

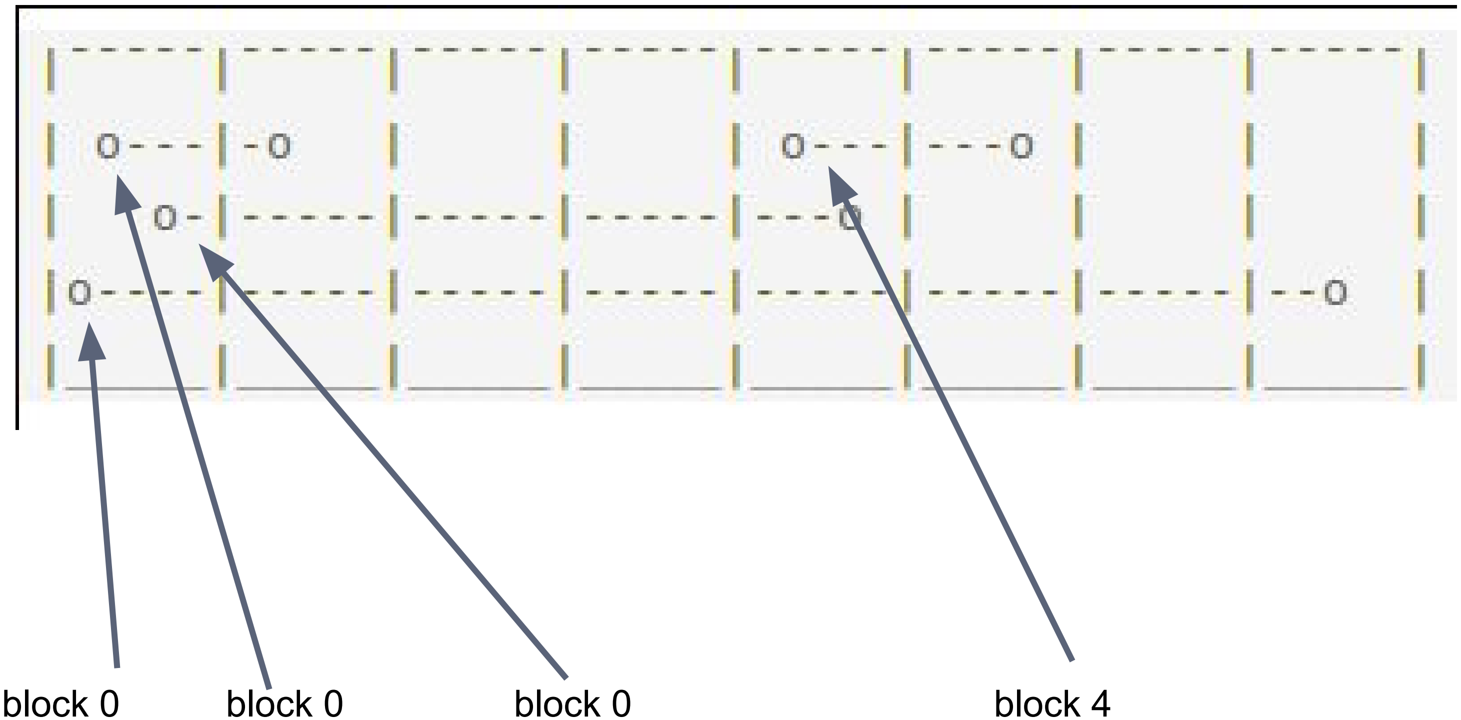
  - Order the queries based on
  - 1) The block idx of the query (smaller first)
  - 2) If tie, the smaller **query Right first**
- Once we ordered, do the normal processing
  - Which equals to = process queries block by block

MO's Algorithm



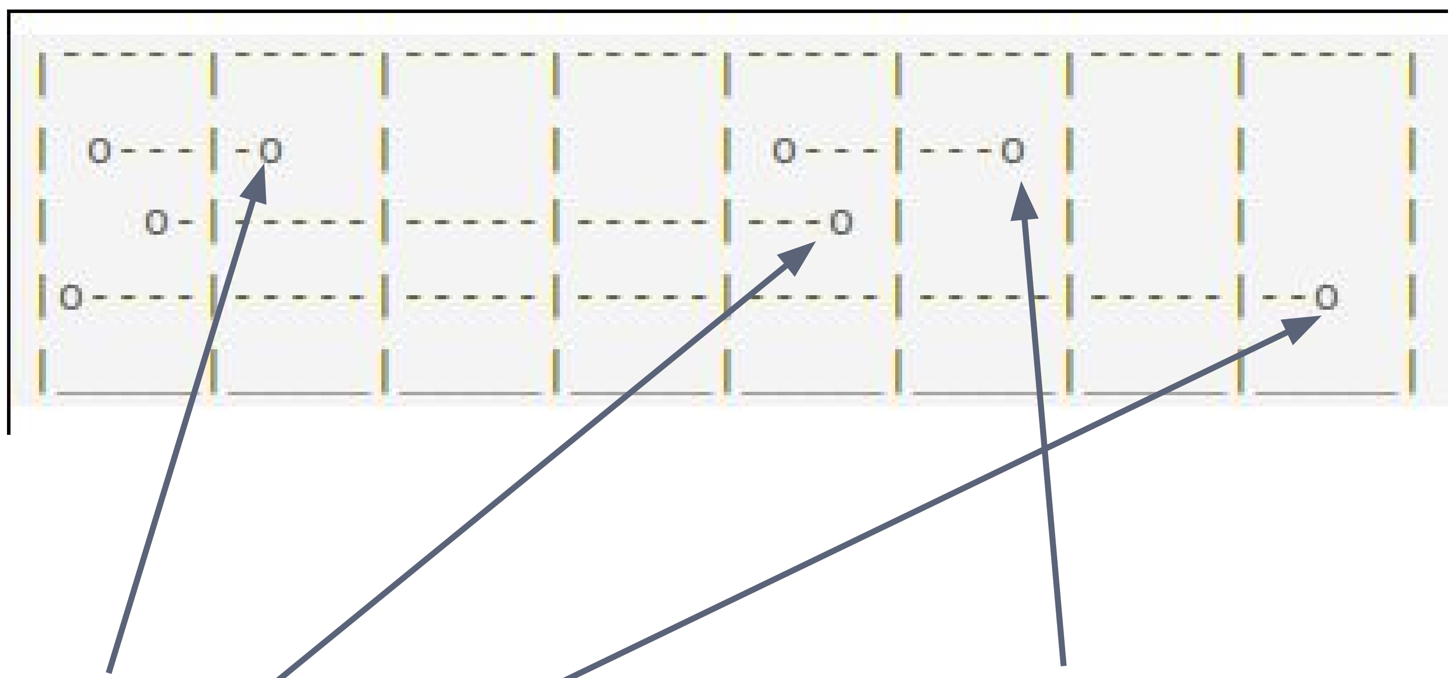
Src: https://www.hackerearth.com/practice/notes/mos-algorithm/

MO’s Algorithm (order on block)



Src: https://www.quora.com/How-exactly-is-the-square-root-decomposition-of-queries-also-sometimes-referred-to-as-Mos-Algorithm-used-for-offline-processing-of-queries

MO's Algorithm (then on Q.right)



block 0
1st query

block 0
2nd query

block 0
3rd query

block 4
4th query

Src: https://www.quora.com/How-exactly-is-the-square-root-decomposition-of-queries-also-sometimes-referred-to-as-Mos-Algorithm-used-for-offline-processing-of-queries

MO’s Algorithm ordering example

- Let's have 3 blocks each of size 3.
  - {0, 3} {1, 7} {2, 8} {7, 8} {4, 8} {4, 4} {1, 2}
  - In terms of blocks indices (based on left):
  - block 0 [0, 1, 2], block 1 [3, 4, 5], block 2 [6, 7, 8]
  - So queries are in blocks: 0 0 0 2 1 1 0
  - Let us re-order them based on their block number.
  - {0, 3} {1, 7} {2, 8} {1, 2} {4, 8} {4, 4} {7, 8}
  - Now let us re-order ties based on their R value.
  - **{1, 2} {0, 3} {1, 7} {2, 8}** {4, 4} {4, 8} {7, 8}

Complexity

- Interesting and challenging
  - Look from **currentR** perspective first
  - For each block, the queries are sorted in increasing order
  - then currentR moves in **increasing order**
  - worst case: currentR will move to end of array = O(N)
  - So each block, might needs O(N)
  - Recall: we have sqrt(N) blocks
  - So currentR moves are: O(NSqrt(N))

Complexity

- Interesting and challenging
  - Look from **currentL** perspective first
  - By definition, a block has some queries start in it
  - When left update from a **query to another**, it navigates in the same block only, not whole array
  - So, each query might does: O(Sqrt(N))
  - For total Q queries: O(Q * Sqrt(N))

Complexity

- Interesting and challenging
  - Navigating from block to next one
  - At sometime, we finish a block queries and move to another
  - In worst case, this navigation is O(N)
  - So total O(N Sqrt(N))
- Total for all cases: O((N+Q)Sqrt(N))

Code: Don’t Change Struct

```
const int INP_SIZE = 30000+9;
const int QUERIES_SIZE = 200000+9;
const int SQRTN = 175;  // sqrt(INP_SIZE)

struct query {
  int l, r, q_idx, block_idx;

  query() {}
  query(int _l, int _r, int _q_idx) {
    l = _l - 1, r = _r - 1, q_idx = _q_idx, block_idx = _l / SQRTN;
  }
  bool operator <(const query &y) const {
    if (block_idx != y.block_idx)
      return block_idx < y.block_idx;
    return r < y.r;
  }
};
```

Code: To change

```
int n, m; // input size and queries
int inp[INP_SIZE], result = 0;
int q_ans[QUERIES_SIZE];
query queries[QUERIES_SIZE];

// You need to update following data structure
  // per problem (e.g. use mutliset)
int cnt[1000000 + 9];

// You need to update these 2 methods per a problem
void add(int idx) {
  cnt[inp[idx]]++;
  if (cnt[inp[idx]] == 3)
    result++;
}

void remove(int idx) {
  cnt[inp[idx]]--;
  if (cnt[inp[idx]] == 2)
    result--;
}
```

Code: Don’t Change

```
void process() {  // don't change
  sort(queries, queries+m);

  int curL = 1, curR = 0; // tricky initialization and indexing
  for (int i = 0; i < m; i++) {
    while (curL < queries[i].l) remove(curL++);
    while (curL > queries[i].l) add(--curL);
    while (curR < queries[i].r) add(++curR);
    while (curR > queries[i].r) remove(curR--);
    q_ans[queries[i].q_idx] = result;
  }
}
```

Code: Main

```
scanf("%d", &n);
for (int i = 0; i < n; i++)
  scanf("%d", &inp[i]);

scanf("%d", &m);
for (int i = 0; i < m; i++) {
  int left, right;
  scanf("%d%d", &left, &right);
  queries[i] = query(left, right, i);
}
process();

for (int i = 0; i < m; i++)
  printf("%d\n", q_ans[i]);
```

Code

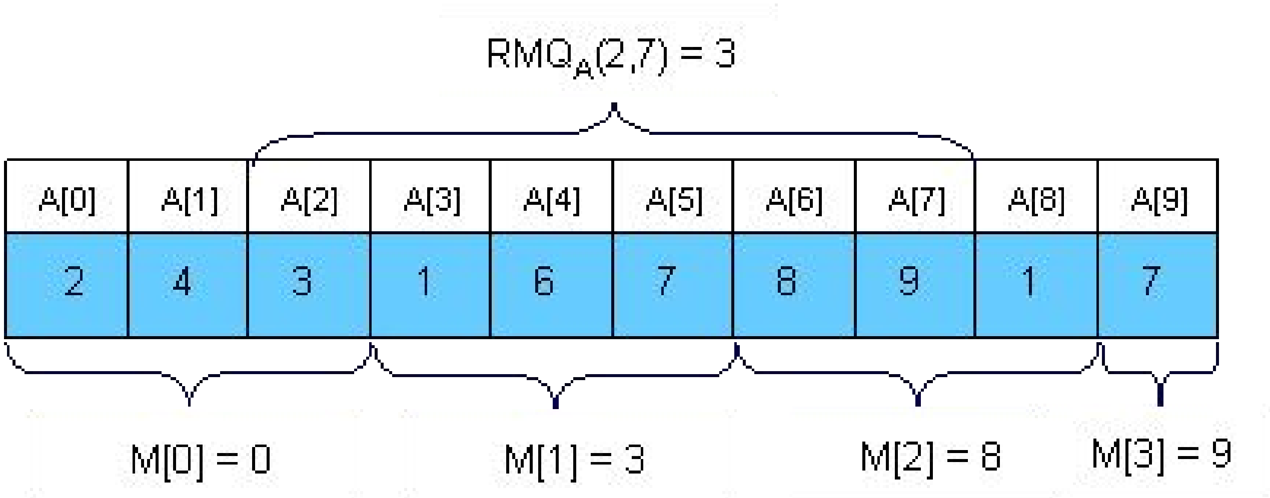
- Java Code
- From a problem to another, you only need to provide **add/remove operations** and its associated data structure (array, set, bbst…)

MO's Algorithm

- Works well for N around 10000
  - Hence total around 1M operations
  - If N is much big (e.g. 10^5) and have no other ideas?
  - Code MO, try **different block sizes**, and **pray for AC**
- Major condition
  - Compute range anwer from an overlapping range query
  - It won't work if **queries order** matter
  - It won't work for **update operations**
- 2D queries: Most probably not a good idea

Other sqrt decomposition Algo

- There is another nice sqrt decomposition Algo
  - Please watch the end of segment tree video
  - You may also read from: TC. For code & explantation
  - Please revise. It fits more with **update operations**.



Sqrt decomposition Algorithms

- MO’s Algorithm
  - Add/Remove operations are need.
  - **Condition**: Efficiently, compute a range from another
  - O((N+Q)Sqrt(N)) for O(1) add/Remove
  - offline processing and doesn’t support update queries
- Other decomp Algorithm
  - Add operations are need.
  - **Condition**: Efficiently, merge results of 2 blocks
  - O(QSqrt(N)) for O(1) add [preprocessing is O(n)]
- 2D: both are **not** good for high dimensions
- MO’s is much more applicable

Sqrt decomposition Algorithms

- Today problem (Frequency >= 3)
  - MO's is O(1) add/remove. Total O((N+Q)Sqrt(N))
  - Other decomp: needs O(Values) memory per a block. needs O(N) time for merging results in array per query. So totally impractical!
- Range Min Query
  - MO's need to replace array with multiset. Extra O(logn) multiplied to add/remove in it. Now, more slower!
  - Other decomp: Need to maintain 1 value only per block, the min value. Merging 2 blocks is O(1). Total O(QSqrt(N)), efficient for N ~= 30000.

Sqrt decomposition Algorithms

- What if we have update operations?
- Today problem (Frequency >= 3)
  - MO's fails. The other one already is not practical.
- Range Min Query
  - MO's fails (in update operations anyway).
  - Other decomp: Need to maintain a multiset per a block. O(log(N)) for update, but still O(1) for merge. In 1st/last query's block, we can use other normal array to iterate.
  - Let Q1 be update queries, Q2 be remaining queries
  - Hence, update queries are O(Q1Log(N))
  - Request queries: O(Q2Sqrt(N)), still efficient

Sqrt(N) vs log(N)

- Why sqrt(N) not other values?
  - Sqrt(N) implies 2 things
  - **We have Sqrt(N) blocks**
  - **Each block is Sqrt(N) numbers**
  - So it helps the complexity we showed
  - What about block size = log(N)?
  - Recall log(N) << sqrt
  - Now: # of blocks = N/log(N).
  - Your turn, recompute the 3 sub-complexities and see how total complexity go up

Mo's Algorithm on Trees

- What if the query on a **tree**?
  - The idea is to convert the tree to array
  - Query 1: Compute something on subtree of given **root**?
  - Easy: Preorder traversal over tree flats it to array. Then any root can be identified in this array as a range
  - Query 2: Given Node A, Node B, query on path?
  - Similar to LCA, run Euler walk DFS to flat a tree to array
  - Then path is (A, B) can be mapped to a range in the array
  - After that, just apply MO if applicable
  - Tutoiral in case

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً