← Notes

9 LIVE EVENTS

Mo's algorithm

74 Mo's algorithm Mo algorithm Algorithm Code Monk CodeMonk

Introduction

Mo's algorithm is a generic idea. It applies to the following class of problems:

> You are given array **Arr** of length **N** and **Q** queries. Each query is represented by two numbers L and R, and it asks you to compute some function **Func** with subarray Arr[L..R] as its argument.

For the sake of brevity we will denote Func([L, R]) as the value of Func on subarray Arr[L..R].
If this sounds too abstract, let's look at specific example:

> There is an integer array Arr of length N and Q queries. For each i, query #i asks you to output the sum of numbers on subarray $[L_i, R_i]$, i.e. $Arr[L_i] + Arr[L_i + 1] + ... + Arr[R_i]$.

Here we have Func([L, R]) = Arr[L] + Arr[L + 1] + ... + Arr[R].
This does not sound so scary, does it? You've probably heard of solutions to this problem using Segment Trees or Binary Indexed Trees, or even prefix sums.

Mo's algorithm provides a way to **answer all queries in O((N + Q) * sqrt(N) * F) time** with at least O(Q) additional memory. Meaning of F is explained below.

The algorithm is applicable if all following conditions are met:

1. Arr is not changed by queries;
2. All queries are known beforehand (techniques requiring this property are often called "offline algorithms");
3. If we know Func([L, R]), then we can compute Func([L + 1, R]), Func([L - 1, R]), Func([L, R + 1]) and Func([L, R - 1]), each in **O(F)** time.

Due to constraints on array immutability and queries being known, Mo's algorithm is inapplicable most of the time. Thus, knowing the method can help you on rare occasions. But. Due to property #3, the algorithm can solve problems that are unsolvable otherwise. If the problem was meant to be solved using Mo's algorithm, then you can be 90% sure that it can not be accepted without knowing it. Since the approach is not well-known, in situations where technique is appropriate, you will easily overcome majority of competitors.

Basic overview

We have Q queries to answer. Suppose we answer them in order they are asked in the following manner:

```
for i = 0..Q-1:
  L, R = query #i
```

```
  for j = L..R:
    do some work to compute Func([L, R])
```

This can take Ω(N * Q) time. If N and Q are of order 10^5, then this would lead to time limit exceeded.

But what if we answer queries in different order? Can we do better then?

Definition #1:
Segment [L, R] is a continuous subarray Arr[L..R], i.e. array formed by elements Arr[L], Arr[L + 1], ..., Arr[R]. We call L **left endpoint** and R **right endpoint** of segment [L, R]. We say that index i belongs to segment [L, R] if L ≤ i ≤ R.

Notation

1. Throughout this tutorial "$^x/_y$" will mean "integer part of x divided by y". For instance, $^{10}/_4 = 2$, $^{15}/_3 = 5$, $^{27}/_8 = 3$;
2. By "sqrt(x)" we will mean "largest integer less or equal to square root of x". For example, sqrt(16) = 4, sqrt(39) = 6;
3. Suppose a query asks to calculate Func([L, R]). We will denote this query as [L, R] - the same way as the respective argument to Func;
4. Everything is 0-indexed.

We will describe Mo's algorithm, and then prove its running time.

The approach works as follows:

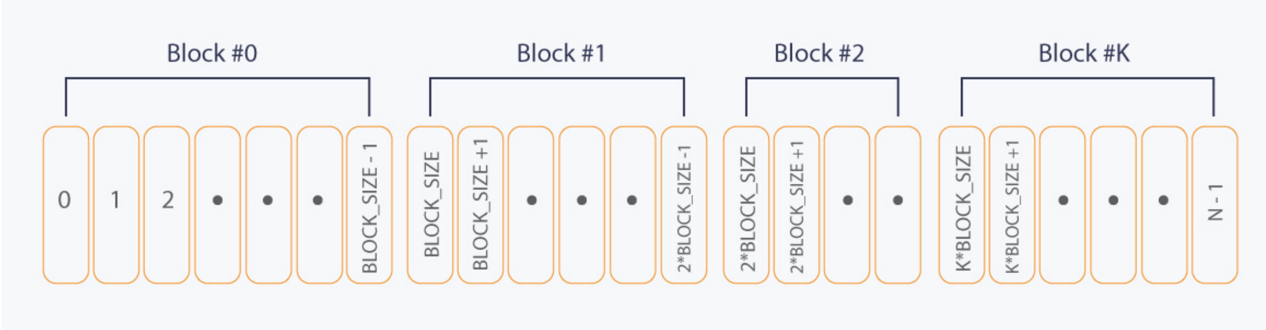
1. Denote BLOCK_SIZE = sqrt(N);
2. Rearrange all queries in a way we will call **"Mo's order"**. It is defined like this: $[L_1, R_1]$ comes earlier than $[L_2, R_2]$ in Mo's order if and only if:

   a) $^{L_1}/_{BLOCK\_SIZE} < {}^{L_2}/_{BLOCK\_SIZE}$

   b) $^{L_1}/_{BLOCK\_SIZE} == {}^{L_2}/_{BLOCK\_SIZE}$ && $R_1 < R_2$
3. Maintain segment [mo_left, mo_right] for which we know Func([mo_left, mo_right]). Initially, this segment is empty. We set mo_left = 0 and mo_right = -1;
4. Answer all queries following Mo's order. Suppose the next query you want to answer is [L, R]. Then you perform these steps:
   a) while mo_right is less than R, extend current segment to [mo_left, mo_right + 1];
   b) while mo_right is greater than R, cut current segment to [mo_left, mo_right - 1];
   c) while mo_left is greater than L, extend current segment to [mo_left - 1, mo_right];
   d) while mo_left is less than L, cut current segment to [mo_left + 1, mo_right].
   This will take O((|left - L| + |right - R|) * F) time, because we required that each extension\deletion is performed in O(F) steps. After all transitions, you will have mo_left = L and mo_right = R, which means that you have successfully computed Func([L, R]).

Time complexity

Let’s view Arr as a union of disjoint segments of size BLOCK_SIZE, which we will call “**blocks**”. Take a look at the picture for better understanding:



Let K be the index of last block. Then there are K + 1 blocks, because we number them from zero. Notice than K’th block can have less than BLOCK_SIZE elements.

Proposition #1:
K = O(sqrt(N)).
Proof:
If sqrt(N) * sqrt(N) = N (which may be false due to our definition of sqrt(N)), then K = sqrt(N) - 1, because we have K + 1 blocks, each of size sqrt(N). Otherwise, K = sqrt(N), because we need one additional block to store N - sqrt(N) * sqrt(N) elements.

Proposition #2:
Block #r is a segment [r * BLOCK_SIZE, min(N - 1, (r + 1) * BLOCK_SIZE - 1)].
Proof (by induction):
For block #0 statement is true — it is a segment [0, BLOCK_SIZE - 1], containing BLOCK_SIZE elements.
Suppose first T ≤ K blocks satisfy the above statement. Then the last of those blocks is a segment [(T - 1) * BLOCK_SIZE, T * BLOCK_SIZE - 1].
Then we form the next, T+1’th block. First element of this block will have array index T * BLOCK_SIZE. We have at most BLOCK_SIZE elements to add to block (there may be less if T + 1 = K + 1). So the last index in T+1’th block will be min(N - 1, (T + 1) * BLOCK_SIZE - 1).

Corollary #1: Two indices i and j belong to same block #r if and only if ${}^{i}/_{BLOCK\_SIZE} = {}^{j}/_{BLOCK\_SIZE} = r$.

Definition #2:

Q_r = { query [L, R] | ${}^{L}/_{BLOCK\_SIZE} = r$ }. Informally, Q_r is a set of queries from input, whose left endpoints belong to block #r. Notice that this set may be empty. We denote the size of Q_r as $|Q_r|$.

Proposition #3:
For each r, queries from Q_r lie continuously in Mo’s order and they appear in it in non-decreasing order of right endpoints.
Queries from Q_r come earlier than queries from Q_{r+1} for every r = 0..K-1.
Proof follows from definition of Mo’s order.

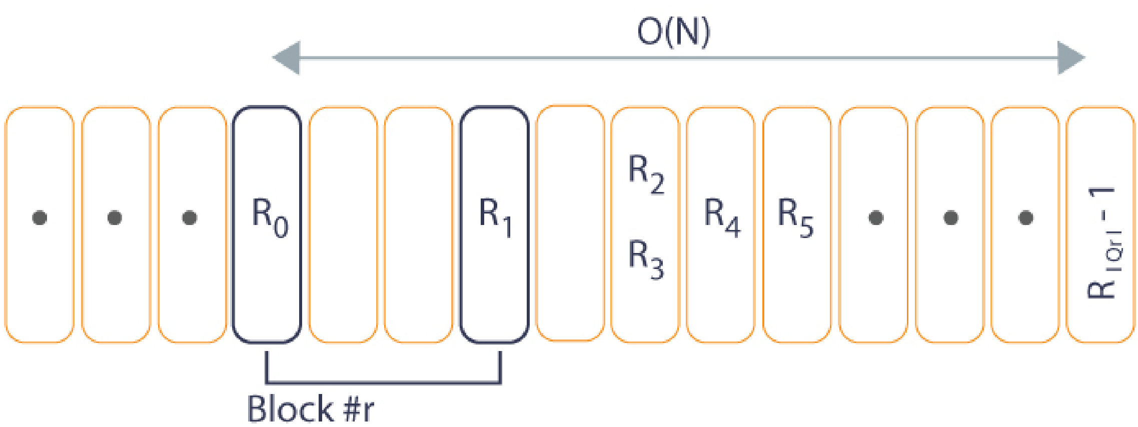
Corollary #2: When we are processing queries following Mo’s order, we firstly process all queries from Q_0, then all queries from Q_1, and so on, up to Q_K.

Theorem #1:

Mo_right changes its value O(N * sqrt(N)) times throughout the run of Mo's algorithm.
Proof:

1. Suppose we've just started processing queries from Q_r for some r, and we've already answered first (following Mo's order) query from it. Let that query be [L, R_0]. This means that now mo_right = R_0. Let R_0, R_1, ..., $R_{|Q_r| - 1}$ be right endpoints of queries from Q_r, in order they appear in Mo's order.

   From proposition #3 we know that
   $R_0 \le R_1 \le R_2 \le \ldots \le R_{|Q_r| - 1}$

   From proposition #2 and definition of Q_r we know that
   r * BLOCK_SIZE ≤ L
   Since right endpoint is not lower that left endpoint (i.e. L ≤ R), we conclude that
   r * BLOCK_SIZE ≤ R_0
   We have N elements in Arr, so any query has right endpoint less than N. Therefore,
   $R_{|Q_r| - 1} \le N - 1$
   Since R's are not decreasing, the total amount of times mo_right changes is
   $R_{|Q_r| - 1} - R_0 \le N - 1 - r * \text{BLOCK\_SIZE} = O(N)$
   (we've substituted $R_{|Q_r| - 1}$ with its highest possible value, and R_0 with its lowest possible value to maximize the subtraction).
   There are O(sqrt(N)) different values of r (proposition #1), so in total we will have O(N * sqrt(N)) changes, assuming that we've already started from the first query of each Q_r.
2. Now suppose we've just ended processing queries from Q_r and we must process first query from $Q_{r + 1}$ (assuming that it is not empty. If it is, then choose next non-empty set). Currently, mo_right is constrained to be:
   r * BLOCK_SIZE ≤ mo_right ≤ N - 1
   Let the first query from $Q_{r + 1}$ be [L', R']. We know (similarly to previous paragraph) that
   (r + 1) * BLOCK_SIZE ≤ R' ≤ N - 1
   Hence, mo_right must be changed at most
   max(|r * BLOCK SIZE - (N - 1)|, N - 1 - (r + 1) * BLOCK SIZE)
   times (we took lowest value of mo_right with highest value of R' and vice-versa). This is clearly O(N) (and it does not matter whether it is r+1'th set of r+k'th for some k > 1).
   There are O(sqrt(N)) switches from r to r + 1 (and it's true even if we skip some empty sets), so in total we will have O(N * sqrt(N)) mo_right changes to do this.

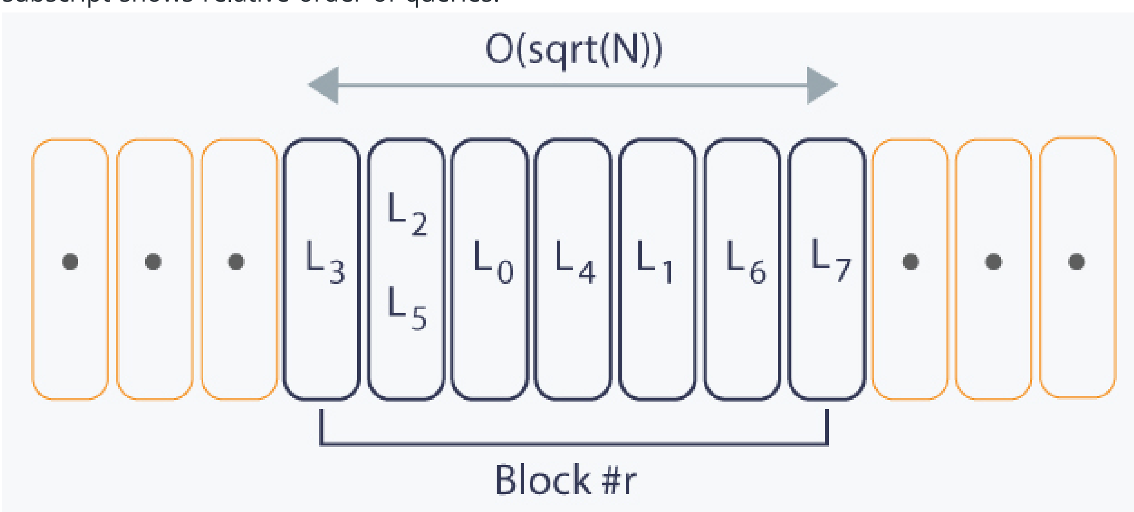
There are no more cases when mo_right changes. Overall, we have O(N * sqrt(N)) + O(N * sqrt(N)) = O(N * sqrt(N)) changes of mo_right.

Corollary #3: All mo_right changes combined take O(N * sqrt(N) * F) time (because each change is done in O(F) time).

Theorem #2:

Mo_left changes its value O(Q * sqrt(N)) times throughout the run of Mo's algorithm.
Proof:

1. Suppose, as in the proof of Theorem #1, we've just started processing queries from Q_r for some r, and currently mo_left = L, mo_right = R_0, where [L, R_0] ∈ Q_r. For every query [L', R'] ∈ Q_r by definition #2 and proposition #2 we have:
   r * BLOCK_SIZE ≤ L' ≤ (r + 1) * BLOCK_SIZE - 1
   So, when we change mo_left from one query to other (both queries ∈ Q_r), we must do at most
   (r + 1) * BLOCK_SIZE - 1 - r * BLOCK_SIZE = BLOCK_SIZE - 1 = O(sqrt(N))
   changes to mo_left. At the picture below we represented leftpoints that lie in block #r, where the subscript shows relative order of queries:



   There are $|Q_r|$ queries in Q_r. That means, that we can estimate upper bound on number of changes for single r as O($|Q_r|$ * sqrt(N)). Let's sum it over all r:
   O(sqrt(N) * ($|Q_0|$ + $|Q_1|$ + ... + $|Q_K|$)) = O(sqrt(N) * Q)
   (because each query's leftpoint belongs to exactly one block, and we have Q queries in total).
2. Now suppose we're done with queries from Q_r and want to process queries from non-empty set Q_{r+k} (for some k > 0). Any query [L', R'] ∈ Q_{r+k} has
   (r + k) * BLOCK_SIZE ≤ L' ≤ (r + k + 1) * BLOCK_SIZE - 1
   Similarly, any query [L, R] ∈ Q_r has
   r * BLOCK_SIZE ≤ L ≤ (r + 1) * BLOCK_SIZE - 1
   We can see that the maximum number of changes needed to transit from some query from Q_r to some query from Q_{r+k} is
   (r + k + 1) * BLOCK_SIZE - 1 - r * BLOCK_SIZE = k * BLOCK_SIZE - 1
   Now if we sum this over all r, we will get at most

K * BLOCK_SIZE = O(sqrt(N)) * O(sqrt(N)) = O(N),
because sum of all possible k's does not exceed K.

There are no more cases when mo_left changes. Overall, we have O(sqrt(N) * Q) + O(N) = O(sqrt(N) * Q) changes of mo_left.

Corollary #4: All mo_left changes combined take O(Q * sqrt(N) * F) time (because each change is done in O(F) time).

Corollary #5: Time complexity of Mo's algorithm is O((N + Q) * sqrt(N) * F).

Example problem

Let's look at an example problem (idea taken from here):

> You have an array Arr of N numbers ranging from 0 to 99. Also you have Q queries [L, R]. For each such query you must print
>
> V([L, R]) = $\sum_{i=0..99}$ count(i)^2 * i
>
> where count(i) is the number of times i occurs in Arr[L..R].

Constraints are N ≤ 10^5, Q ≤ 10^5.
To apply Mo's algorithm, you must ensure of three properties:

1. Arr is not modified by queries;
2. Queries are known beforehand;
3. If you know V([L, R]), then you can compute V([L + 1, R]), V([L - 1, R]), V([L, R - 1]) and V([L, R + 1]), each in O(F) time.

First two properties obviously hold. The third property depends on the time bound — O(F).
Surely, we can compute V([L + 1, R]) from scratch in Ω(R - L) = Ω(N) in the worst case. But looking at complexity of Mo's algorithm, you can deduce that this will surely time out, because we multiply O(F) with O((Q + N) * sqrt(N)).
Typically, you want O(F) to be O(1) or O(log(n)). **Choosing a way to achieve appropriate time bound O(F) is programmer's main concern when solving problems using Mo's algorithm.**

I will describe a way to achieve O(F) = O(1) for this problem.
Let's maintain variable **current_answer** (initialized by zero) to store V([mo_left, mo_right]) and integer array **cnt** of size 100, where cnt[x] will be the number of times x occurs in [mo_left, mo_right].
From the definition of current_answer we can see that

current answer = V([mo_left, mo_right]) =$\sum_{i=0..99}$ count'(i)^2 * i

where count'(x) is the number of times x occurs in [mo_left, mo_right].
By the definition of cnt we see that count'(x) = cnt[x], so

current answer = $\sum_{i=0..99}$ cnt[i]^2 * i.

Suppose we want to change mo_right to mo_right + 1. It means we want to add number p = Arr[mo_right + 1] into consideration. But we also want to retain current_answer and cnt's properties.
Maintaining cnt is easy: just increase cnt[p] by 1. It is a bit trickier to deal with current_answer.
We know (from the definitions) that current_answer contains summand cnt[p]^2 * p before addition. Let's

subtract this value from the answer. Then, after we perform addition to cnt, add again $cnt[p]^2 * p$ to the answer (make no mistake: this time it will contain updated value of cnt[p]). Both updates take O(1) time. All other transitions (mo_left to mo_left + 1, mo_left to mo_left - 1 and mo_right to mo_right - 1) can be done in the same way, so we have O(F) = O(1). You can refer to the code below for clarity.

Now let's look into detail on one sample test case:
Input:
Arr = [0, 1, 1, 0, 2, 3, 4, 1, 3, 5, 1, 5, 3, 5, 4, 0, 2, 2] of 18 elements
Queries (0-indexed): [0, 8], [2, 5], [2, 11], [16, 17], [13, 14], [1, 17], [17, 17]

The algorithm works as follows:
Firstly, set BLOCK_SIZE = sqrt(18) = 4. Notice that we have 5 blocks: [0, 3], [4, 7], [8, 11], [12, 15], [16, 17]. The last block contains less than BLOCK_SIZE elements.
Then, set mo_left = 0, mo_right = -1, current_answer = 0, cnt = [0, 0, 0, 0, 0, 0] (I will use only first 6 elements out of 100 for the sake of simplicity).

Then sort queries. The Mo's order will be:
[2,5], [0, 8], [2, 11], [1, 17] (here ends Q_0) [13, 14] (here ends Q_3) [16, 17], [17, 17] (here ends Q_4).

Now, when everything is set up, we can answer queries:

1. We need to process query [2, 5]. Currently, our segment is [0, -1]. So we need to move mo_right to 5 and mo_left to 2.
   Let's move mo_right first:
   mo_right = 0, current_answer = 0, cnt = [1, 0, 0, 0, 0, 0]
   mo_right = 1, current_answer = 1, cnt = [1, 1, 0, 0, 0, 0]
   mo_right = 2, current_answer = 4, cnt = [1, 2, 0, 0, 0, 0]
   mo_right = 3, current_answer = 4, cnt = [2, 2, 0, 0, 0, 0]
   mo_right = 4, current_answer = 6, cnt = [2, 2, 1, 0, 0, 0]
   mo_right = 5, current_answer = 9, cnt = [2, 2, 1, 1, 0, 0]
   Now we must move mo_left:
   mo_left = 1, current_answer = 9, cnt = [1, 2, 1, 1, 0, 0]
   mo_left = 2, current_answer = 6, cnt = [1, 1, 1, 1, 0, 0]
   Thus, the answer for query [2, 5] is 6.
2. Our next query is [0, 8]. Current segment [mo_left, mo_right] is [2, 5]. We need to move mo_right to 8 and mo_left to 0.
   Again, let's move mo_right first:
   mo_right = 6, current_answer = 10, cnt = [1, 1, 1, 1, 1, 0]
   mo_right = 7, current_answer = 13, cnt = [1, 2, 1, 1, 1, 0]
   mo_right = 8, current_answer = 22, cnt = [1, 2, 1, 2, 1, 0]
   Then we move mo_left:
   mo_left = 1, current_answer = 27, cnt = [1, 3, 1, 2, 1, 0]
   mo_left = 0, current_answer = 27, cnt = [2, 3, 1, 2, 1, 0]
   So, the answer for query [0, 8] is 27.
3. Next query is [2, 11]. Current segment is [0, 8]. We need to move mo_right to 11 and mo_left to 2.
   mo_right = 9, current_answer = 32, cnt = [2, 3, 1, 2, 1, 1]
   mo_right = 10, current_answer = 39, cnt = [2, 4, 1, 2, 1, 1]
   mo_right = 11, current_answer = 54, cnt = [2, 4, 1, 2, 1, 2]

mo_left = 1, current_answer = 54, cnt = [1, 4, 1, 2, 1, 2]
mo_left = 2, current_answer = 47, cnt = [1, 3, 1, 2, 1, 2]
Answer for query [2, 11] is 47.

4. Next query is [1, 17]. Current segment is [2, 11]. We need to move mo_right to 17 and mo_left to 1.
   mo_right = 12, current_answer = 62, cnt = [1, 3, 1, 3, 1, 2]
   mo_right = 13, current_answer = 87, cnt = [1, 3, 1, 3, 1, 3]
   mo_right = 14, current_answer = 99, cnt = [1, 3, 1, 3, 2, 3]
   mo_right = 15, current_answer = 99, cnt = [2, 3, 1, 3, 2, 3]
   mo_right = 16, current_answer = 105, cnt = [2, 3, 2, 3, 2, 3]
   mo_right = 17, current_answer = 115, cnt = [2, 3, 3, 3, 2, 3]
   mo_left = 1, current_answer = 122, cnt = [2, 4, 3, 3, 2, 3]
   Answer for query [1, 17] is 122.
5. Our next goal is query [13, 14]. Notice that it starts in different block from the previous query [1, 17]. Consequently, this is the first time mo_right will move to the left. We need to move mo_left to 13 and mo_right to 14.
   mo_right = 16, current_answer = 112, cnt = [2, 4, 2, 3, 2, 3]
   mo_right = 15, current_answer = 106, cnt = [2, 4, 1, 3, 2, 3]
   mo_right = 14, current_answer = 106, cnt = [1, 4, 1, 3, 2, 3]
   mo_left = 2, current_answer = 99, cnt = [1, 3, 1, 3, 2, 3]
   mo_left = 3, current_answer = 94, cnt = [1, 2, 1, 3, 2, 3]
   mo_left = 4, current_answer = 94, cnt = [0, 2, 1, 3, 2, 3]
   mo_left = 5, current_answer = 92, cnt = [0, 2, 0, 3, 2, 3]
   mo_left = 6, current_answer = 77, cnt = [0, 2, 0, 2, 2, 3]
   mo_left = 7, current_answer = 65, cnt = [0, 2, 0, 2, 1, 3]
   mo_left = 8, current_answer = 62, cnt = [0, 1, 0, 2, 1, 3]
   mo_left = 9, current_answer = 53, cnt = [0, 1, 0, 1, 1, 3]
   mo_left = 10, current_answer = 28, cnt = [0, 1, 0, 1, 1, 2]
   mo_left = 11, current_answer = 27, cnt = [0, 0, 0, 1, 1, 2]
   mo_left = 12, current_answer = 12, cnt = [0, 0, 0, 1, 1, 1]
   mo_left = 13, current_answer = 9, cnt = [0, 0, 0, 0, 1, 1]
   Answer for query [13,14] is 9.
6. Next query is [16, 17]. Notice, however, that now we do not need to move mo_right to the left. We need to move mo_left to 16 and mo_right to 14.
   mo_right = 15, current_answer = 9, cnt = [1, 0, 0, 0, 1, 1]
   mo_right = 16, current_answer = 11, cnt = [1, 0, 1, 0, 1, 1]
   mo_right = 17, current_answer = 17, cnt = [1, 0, 2, 0, 1, 1]
   mo_left = 14, current_answer = 12, cnt = [1, 0, 2, 0, 1, 0]
   mo_left = 15, current_answer = 8, cnt = [1, 0, 2, 0, 0, 0]
   mo_left = 16, current_answer = 8, cnt = [0, 0, 2, 0, 0, 0]
   Answer for [16, 17] is 8.
7. The last query is [17, 17]. It requires us to move mo_left one unit to the right.
   mo_left = 17, current_answer = 2, cnt = [0, 0, 1, 0, 0, 0]
   Answer for this query is 2.

Now the important part comes: we must **output answers** not in order we've achieved them, but in **order they were asked**.

Output:
27
6
47
8
9
122
2

Implementation

Here is the C++ implementation for the above problem:

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

int N, Q;

// Variables, that hold current "state" of computation
long long current_answer;
long long cnt[100];

// Array to store answers (because the order we achieve them is messed up)
long long answers[100500];
int BLOCK_SIZE;
int arr[100500];

// We will represent each query as three numbers: L, R, idx. Idx is
// the position (in original order) of this query.
pair< pair<int, int>, int> queries[100500];



// Essential part of Mo's algorithm: comparator, which we will
// use with std::sort. It is a function, which must return True
// if query x must come earlier than query y, and False otherwise.
inline bool mo_cmp(const pair< pair<int, int>, int> &x,
        const pair< pair<int, int>, int> &y)
{
    int block_x = x.first.first / BLOCK_SIZE;
    int block_y = y.first.first / BLOCK_SIZE;
    if(block_x != block_y)
        return block_x < block_y;
    return x.first.second < y.first.second;
}

// When adding a number, we first nullify it's effect on current
// answer, then update cnt array, then account for it's effect again.
```

```
inline void add(int x)
{
    current_answer -= cnt[x] * cnt[x] * x;
    cnt[x]++;
    current_answer += cnt[x] * cnt[x] * x;
}

// Removing is much like adding.
inline void remove(int x)
{
    current_answer -= cnt[x] * cnt[x] * x;
    cnt[x]--;
    current_answer += cnt[x] * cnt[x] * x;
}

int main()
{
    cin.sync_with_stdio(false);
    cin >> N >> Q;
    BLOCK_SIZE = static_cast<int>(sqrt(N));

    // Read input array
    for(int i = 0; i < N; i++)
        cin >> arr[i];

    // Read input queries, which are 0-indexed. Store each query's
    // original position. We will use it when printing answer.
    for(int i = 0; i < Q; i++) {
        cin >> queries[i].first.first >> queries[i].first.second;
        queries[i].second = i;
    }

    // Sort queries using Mo's special comparator we defined.
    sort(queries, queries + Q, mo_cmp);

    // Set up current segment [mo_left, mo_right].
    int mo_left = 0, mo_right = -1;

    for(int i = 0; i < Q; i++) {
        // [left, right] is what query we must answer now.
        int left = queries[i].first.first;
        int right = queries[i].first.second;

        // Usual part of applying Mo's algorithm: moving mo_left
        // and mo_right.
        while(mo_right < right) {
            mo_right++;
```

```
            add(arr[mo_right]);
        }
        while(mo_right > right) {
            remove(arr[mo_right]);
            mo_right--;
        }

        while(mo_left < left) {
            remove(arr[mo_left]);
            mo_left++;
        }
        while(mo_left > left) {
            mo_left--;
            add(arr[mo_left]);
        }

        // Store the answer into required position.
        answers[queries[i].second] = current_answer;
    }

    // We output answers *after* we process all queries.
    for(int i = 0; i < Q; i++)
        cout << answers[i] << "\n";
    return 0;
}
```

Same solution without global variables (the way I like to implement it):

```
#include <bits/stdc++.h>
using std::vector;
using std::tuple;

/*
 * Take out adding\removing logic into separate class.
 * It provides functions to add and remove numbers from
 * our structure, while maintaining cnt[] and current_answer.
 *
 */
struct Mo
{
    static constexpr int MAX_VALUE = 1005000;
    vector<long long> cnt;
    long long current_answer;

    void process(int number, int delta)
    {
```

```
        current_answer -= cnt[number] * cnt[number] * number;
        cnt[number] += delta;
        current_answer += cnt[number] * cnt[number] * number;
    }
public:
    Mo()
    {
        cnt = vector<long long>(MAX_VALUE, 0);
        current_answer = 0;
    }

    long long get_answer() const
    {
        return current_answer;
    }

    void add(int number)
    {
        process(number, 1);
    }

    void remove(int number)
    {
        process(number, -1);
    }
};

int main()
{
    int N, Q, BLOCK_SIZE;
    std::cin.sync_with_stdio(false);
    std::cin >> N >> Q;
    BLOCK_SIZE = static_cast<int>(sqrt(N));

    // No global variables, everything inside.
    vector<int> arr(N);
    vector<long long> answers(Q);
    vector< tuple<int, int, int> > queries;
    queries.reserve(Q);

    for(int i = 0; i < N; i++)
        std::cin >> arr[i];

    for(int i = 0; i < Q; i++) {
        int le, rg;
        std::cin >> le >> rg;
        queries.emplace_back(le, rg, i);
```

```
    }

    // Comparator as a lambda-function, which catches BLOCK_SIZE
    // from the above definition.
    auto mo_cmp = [BLOCK_SIZE](const tuple<int, int, int> &x,
            const tuple<int, int, int> &y) -> bool {
        int block_x = std::get<0>(x) / BLOCK_SIZE;
        int block_y = std::get<0>(y) / BLOCK_SIZE;
        if(block_x != block_y)
            return block_x < block_y;
        return std::get<1>(x) < std::get<1>(y);
    };

    std::sort(queries.begin(), queries.end(), mo_cmp);

    Mo solver;
    int mo_left = 0, mo_right = -1;

    // Usual Mo's algorithm stuff.
    for(const auto &q: queries) {
        int left, right, answer_idx;
        std::tie(left, right, answer_idx) = q;

        while(mo_right < right) {
            mo_right++;
            solver.add(arr[mo_right]);
        }
        while(mo_right > right) {
            solver.remove(arr[mo_right]);
            mo_right--;
        }

        while(mo_left < left) {
            solver.remove(arr[mo_left]);
            mo_left++;
        }
        while(mo_left > left) {
            mo_left--;
            solver.add(arr[mo_left]);
        }

        answers[answer_idx] = solver.get_answer();
    }

    for(int i = 0; i < Q; i++)
        std::cout << answers[i] << "\n";
```

```
    return 0;
}
```

Practice problems

In case you want to try out implementing Mo's technique for yourself, check out these problems:

1. Kriti and her birthday gift
   Difficulty: easy.
   Prerequisites: string hashing.
   Comment: tests for this problem are flawed (you can get **at most 33 out of 100** for this problem), because both setter's and tester's implementations from the editorial have the same bug in hashing function. Can you see what it is?
   Reference implementation: here.
2. SUBSTRINGS COUNT
   Difficulty: easy.
   Prerequisites: string hashing.
   Comment: almost the same as previous problem, but this one asks slightly different question and has well-formed tests.
   Reference implementation: here.
3. Sherlock and inversions
   Difficulty: medium.
   Prerequisites: segment tree\binary indexed tree, coordinate compression.
   Comment: nice and clean problem on Mo's algorithm. Might seem difficult for people not familiar with prerequisites.
   Reference implementation: here.

COMMENTS (40) SORT BY: Relevance

Join Discussion...

Cancel Post

Max (Ang) Li 2 years ago
How do you create such beautiful diagrams? :^o
0 votes • Reply • Message • Permalink

Mike Koltsov Author 2 years ago
HackerEarth staff did this for me
2 votes • Reply • Message • Permalink

Satyarth 10 months ago
can you please help me with a sequence to learn algorithms. That will be really helpful. Thanks in advance
1 vote • Reply • Message • Permalink

Mike Koltsov Author 10 months ago
I think CodeMonk provides reasonable sequence if you don't know where to start. Visit https://www.hackerearth.com/codemonk/
0 votes • Reply • Message • Permalink

Satyarth 10 months ago
thank you so much
0 votes • Reply • Message • Permalink

Arjit Srivastava 2 years ago
Thank you for this amazing post, Mike! :)
1 vote • Reply • Permalink

Viet Nguyenkhanh 2 years ago
thanks you very much, the notes very interesting
1 vote • Reply • Message • Permalink

Anand Hariharan 2 years ago
nice post Mike !
1 vote • Reply • Message • Permalink

Aman Goel 2 years ago
Amazing!
I got to learn Fenwick tree as well because of the last problem
Thanks a lot for such a nice explanation of the algorithm
1 vote • Reply • Message • Permalink

maddela sai karthik 10 months ago
Nice
Post mike
Tnxx
1 vote • Reply • Message • Permalink

Vaibhav Tulsyan 10 months ago
Brilliant explanation!
1 vote • Reply • Message • Permalink

Satyarth 10 months ago
Thank you so much.. This is an amazing post and it helped me cope up with my fear of questions like this
1 vote • Reply • Message • Permalink

maddela sai karthik 9 months ago
i think there's a typo in the theorem 2 proof part 1
this line:
There are |Qr| queries in Qr. That means, that we can estimate upper bound on number of changes for single r as O(|Qr| * N). Let's sum it over all r:
O(N * (|Q0| + |Q1| + ... + |QK|)) = O(sqrt(N) * Q)
the N should be actually sqrt(N) in the left hand side of the equation..
1 vote • Reply • Message • Permalink

Mike Koltsov Author 9 months ago
Thank you, nice catch!
0 votes • Reply • Message • Permalink

maddela sai karthik 9 months ago
Can you write a similar article for heavy light decomposition the proof
Part about it in your free time.
0 votes • Reply • Message • Permalink

Dsrm 9 months ago
@Mike
nothing else

thank you
1 vote • Reply • Message • Permalink

Vaibhav Goyal Edited 6 months ago
Nice work Mike :)
you have done a lot of hard work to prepare this tutorial thank you so much :)
and i would like if you will write more on other topics too :)
1 vote • Reply • Message • Permalink

Arun Prasad 2 years ago
Very Good Article.
What is the difference between Mo's Algorithm adn Square Root decompostion ?
0 votes • Reply • Message • Permalink

Mike Koltsov Author 2 years ago
My opinion on this matter: Mo's algorithm is one of the tricks of sqrt decomposition. Some people think that Mo's algorithm can not be distinguished from sqrt decomposition, but I think Mo's algorithm is very interesting itself.
0 votes • Reply • Message • Permalink

Abhishek Bind 2 years ago
Very nice Article.
However, I wanted to point out that in the first implementation while processing queries, left and right should be decreased by 1 for correct indexing.
Thanks :)
0 votes • Reply • Message • Permalink

Mike Koltsov Author 2 years ago
Well, I purposely assume that input queries are 0-based (it is written at the beginning of the example)
0 votes • Reply • Message • Permalink

Ken Mercado a year ago
What is the variable F in the time complexity analysis?
0 votes • Reply • Message • Permalink

Mike Koltsov Author a year ago
It is explained in the introduction as
"If we know Func([L, R]), then we can compute Func([L + 1, R]), Func([L - 1, R]), Func([L, R + 1]) and Func([L, R - 1]), each in *O(F)* time."
F is a function rather than a variable. It is the tightest upper bound you can prove on time complexity of recomputing Func.
0 votes • Reply • Message • Permalink

Iago Shavidze a year ago
great post,but unfortunately this code gives me time limits on codeforces
http://codeforces.com/contest/86/submission/15758953 this is my code,can anyone help me?
0 votes • Reply • Message • Permalink

Mike Koltsov Author a year ago
Rewriting updates to
```
inline void upd(ll x, ll mult)
{
cur_ans += 2 * mult * cnt[x] * x + x;
cnt[x] += mult;
}
void add(ll x){
upd(x, 1);
}
void removeX(ll x){
upd(x, -1);
}
```
Leads to AC in 4.9sec.

9 LIVE EVENTS

Also you can optimize by memsetting cnt to 0 when you encounter new block (instead of moving R to the left), like this (taken from my code):

```
if(i == 0 or q[i].L / len != q[i - 1].L / len) { // new bucket
memset(cnt, 0, sizeof(cnt));
cur = 0;
for(int j = q[i].L; j <= q[i].R; j++) {
// cout << a[j] << " " << cur;
upd(a[j], 1, cur);
// cout << "->" << cur << endl;
}
L = q[i].L, R = q[i].R;
ans[q[i].i] = cur;
} else {
...
}
```

▲ 0 votes • Reply • Message • Permalink

Sai Teja Utpala 4 months ago

First, most elegant explanation https://www.hackerearth.com/users/misha/"
target="_blank">@Mike Koltsov
and To see why this is an optimization i did mathematical analysis(or at least i believe so) following your prints
It goes this way:
if we optimize this way case '2' in both theorems goes away and we get new factor upon which running time depends and that is
worst case update operations we have by making transition to new block and computing first query in it would be
|(r*block_size)-(N-1)| , and summing this over all this 'r'
=Σ|(r*block_size)-(N-1)| and 'r' goes from 0 to (k=(sqrt(N))
=|(sqrt(N)*block_size)-(sqrt(N)*N)|
=O(N*sqrt(N)-N)
total running time would be O(N*sqrt(N) + Q*sqrt(N) + N*sqrt(N)-N) which better than O(N*sqrt(N) + N*sqrt(N + Q*sqrt(N) + N) by factor of 2N which is crucial for larger N in the problem

correct me if am wrong.?

▲ 0 votes • Reply • Message • Permalink

Mike Koltsov Author a year ago

You can refer to my submissions for optimizations mentioned above:
http://codeforces.com/contest/86/submission/15762653 (your solution with first optimization, runs in 4.9s)
http://codeforces.com/contest/86/submission/13404487 (my solution, both optimizations, runs in 3.2s)

▲ 0 votes • Reply • Message • Permalink

Iago Shavidze a year ago

thanks a lot, i have one question about this algorithm, i did not quite understand why we must sort in this logic the queries.

▲ 0 votes • Reply • Message • Permalink

Mike Koltsov Author a year ago

Because it leads to better runtime complexity, compared to naive approach.

▲ 0 votes • Reply • Message • Permalink

Iago Shavidze a year ago

thanks,now understand.

▲ 0 votes • Reply • Message • Permalink

shavidze shavidze a year ago

my last optimizations
http://codeforces.com/contest/86/submission/15895538

▲ 0 votes • Reply • Message • Permalink

Mike Koltsov Author a year ago

Wow, really nice!
0 votes • Reply • Message • Permalink

shavidze shavidze a year ago
mike please, if u can explain me tarjan lca algorithm or give me resources about it ,i search with google tarjan lca algorithm but unfrotunately i can;t understand it , because every explanation is very short and is not clear.
0 votes • Reply • Message • Permalink

Mike Koltsov Author a year ago
If I was to search for Tarjan explanation, I would definitely visit http://e-maxx.ru/algo/lca_linear_offline. It is in russian, though. Maybe Google Translate can help.
0 votes • Reply • Message • Permalink

shavidze shavidze a year ago
thank you very much!
0 votes • Reply • Message • Permalink

Arpan Mukherjee a year ago
WOW! Really nice post,beautifully explained. Thank you. Just learnt the MO's algorithm. Just have one question.
is Square root decomposition and MO' Algorithm same or different? I mean what's the difference between these two(If they're different).
0 votes • Reply • Message • Permalink

Mike Koltsov Author a year ago
Thank you for your kind words!
In my opinion, Mo's algorithm is one example of a very broad topic, which is Square root decomposition.
0 votes • Reply • Message • Permalink

Arpan Mukherjee a year ago
Okay.. Thank you :)
0 votes • Reply • Message • Permalink

Dsrm 9 months ago
what is the use of blocks here except sorting in "mo order"
0 votes • Reply • Message • Permalink

Mike Koltsov Author 9 months ago
Length of all blocks is chosen in such a way, that we can prove O(N * sqrt(N)) time complexity. You can vary it and achieve different results.
What else do you expect or need to know about blocks?
0 votes • Reply • Message • Permalink

AUTHOR

Mike Koltsov
Saint Petersburg, Russia
3 notes

TRENDING NOTES

Python Diaries Chapter 3 Map | Filter | For-else | List Comprehension written by Divyanshu Bansal

Bokeh | Interactive Visualization Library | Use Graph with Django Template
written by Prateek Kumar

Bokeh | Interactive Visualization Library | Graph Plotting
written by Prateek Kumar

Python Diaries chapter 2
written by Divyanshu Bansal

Python Diaries chapter 1
written by Divyanshu Bansal

more ...

9 LIVE EVENTS

About Us | Hackathons | Talent Solutions
University Program | Developers Wiki | Blog
Press | Careers | Reach Us