MO's Algorithm

Ashish Agarwal

December 2020

Problem Statement

Given a sequence of n numbers $A_1, A_2, \ldots, A_n$ and a number of $d - queries$. A $d - query$ is a pair $(l, r)(1 \leq l \leq r \leq n)$. For each $d - query(l, r)$, you have to return the number of distinct elements in the subsequence $A_l, A_{l+1}, \ldots, A_r$.

Example

Let $A = \{1, 1, 2, 1, 3\}$

Example

Let $A = \{1, 1, 2, 1, 3\}$

- Query: $(l, r) = (1, 5)$, $A_l, A_{l+1}, \ldots, A_r = \{1, 1, 2, 1, 3\}$,

Example

Let $A = \{1, 1, 2, 1, 3\}$

- Query: $(l, r) = (1, 5)$, $A_l, A_{l+1}, \ldots, A_r = \{1, 1, 2, 1, 3\}$,
- Answer $= 3$

Example

Let $A = \{1, 1, 2, 1, 3\}$

- Query: $(l, r) = (1, 5)$, $A_l, A_{l+1}, \ldots, A_r = \{1, 1, 2, 1, 3\}$,
- Answer $= 3$
- Query: $(l, r) = (2, 4)$, $A_l, A_{l+1}, \ldots, A_r = \{1, 2, 1\}$

Example

Let $A = \{1, 1, 2, 1, 3\}$

- Query: $(l, r) = (1, 5)$, $A_l, A_{l+1}, \ldots, A_r = \{1, 1, 2, 1, 3\}$,
- Answer $= 3$
- Query: $(l, r) = (2, 4)$, $A_l, A_{l+1}, \ldots, A_r = \{1, 2, 1\}$
- Answer $= 2$

Example

Let $A = \{1, 1, 2, 1, 3\}$

- Query: $(l, r) = (1, 5)$, $A_l, A_{l+1}, \ldots, A_r = \{1, 1, 2, 1, 3\}$,
- Answer $= 3$
- Query: $(l, r) = (2, 4)$, $A_l, A_{l+1}, \ldots, A_r = \{1, 2, 1\}$
- Answer $= 2$
- Query: $(l, r) = (3, 5)$, $A_l, A_{l+1}, \ldots, A_r = \{2, 1, 3\}$

Example

Let $A = \{1, 1, 2, 1, 3\}$

- Query: $(l, r) = (1, 5)$, $A_l, A_{l+1}, \ldots, A_r = \{1, 1, 2, 1, 3\}$,
- Answer $= 3$
- Query: $(l, r) = (2, 4)$, $A_l, A_{l+1}, \ldots, A_r = \{1, 2, 1\}$
- Answer $= 2$
- Query: $(l, r) = (3, 5)$, $A_l, A_{l+1}, \ldots, A_r = \{2, 1, 3\}$
- Answer $= 3$

Naive Solution

```
for each query:
    answer = 0
    freq[] = 0
    for i in {l...r}:
        freq[A[i]]++
        if freq[A[i]] = 1:
            answer++
```

Naive Solution

```
for each query:
    answer = 0
    freq[] = 0
    for i in {l...r}:
        freq[A[i]]++
        if freq[A[i]] = 1:
            answer++
```

The time complexity of the above solution is

Naive Solution

```
for each query:
    answer = 0
    freq[] = 0
    for i in {l...r}:
        freq[A[i]]++
        if freq[A[i]] = 1:
            answer++
```

The time complexity of the above solution is $O(n^2)$.

Modified Solution

```
def add(position):
    freq[array[position]]++
    if freq[array[position]] == 1:
        answer++

def remove(position):
    freq[array[position]]--
    if freq[array[position]] == 0:
        answer--

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

Modified Solution

```
for each query:
    while L < currentL :
        currentL--
        add(currentL)
    while currentR < R:
        currentR++
        add(currentR)
    while currentL < L:
        remove(currentL)
        currentL++
    while R < currentR:
        remove(currentR)
        currentR--

    output answer
```

MO's Algorithm

- MO's algorithm is just an order in which we process the queries.

MO's Algorithm

- MO's algorithm is just an order in which we process the queries.
- This is an offline algorithm.

MO's Algorithm

- MO's algorithm is just an order in which we process the queries.
- This is an offline algorithm.
- Let us divide the given input array into $\sqrt{N}$ blocks. Each block will be $N/\sqrt{N} = \sqrt{N}$ size.

MO's Algorithm

- MO's algorithm is just an order in which we process the queries.
- This is an offline algorithm.
- Let us divide the given input array into $\sqrt{N}$ blocks. Each block will be $N/\sqrt{N} = \sqrt{N}$ size.
- Each query has L and R, we will call them opening and closing. Each opening has to fall in one of these blocks. Each closing has to fall in one of these blocks.

MO's Algorithm

- MO's algorithm is just an order in which we process the queries.
- This is an offline algorithm.
- Let us divide the given input array into $\sqrt{N}$ blocks. Each block will be $N/\sqrt{N} = \sqrt{N}$ size.
- Each query has L and R, we will call them opening and closing. Each opening has to fall in one of these blocks. Each closing has to fall in one of these blocks.
- A query belongs to $P'th$ block if the opening of that query fall in $P'th$ block.

MO's Algorithm

- MO's algorithm is just an order in which we process the queries.
- This is an offline algorithm.
- Let us divide the given input array into $\sqrt{N}$ blocks. Each block will be $N/\sqrt{N} = \sqrt{N}$ size.
- Each query has L and R, we will call them opening and closing. Each opening has to fall in one of these blocks. Each closing has to fall in one of these blocks.
- A query belongs to $P'th$ block if the opening of that query fall in $P'th$ block.
- In this algorithm we will process the queries of $1st$ block. Then we process the queries of $2nd$ block and so on.. finally $\sqrt{N}'th$ block. We already have an ordering, queries are ordered in the ascending order of its block. There can be many queries that belong to the same block.

MO's Algorithm

- Let's ignore all other blocks and focus on answering queries of block 1.

MO's Algorithm

- Let's ignore all other blocks and focus on answering queries of block 1.
- We will similarly do for all blocks.

MO's Algorithm

- Let's ignore all other blocks and focus on answering queries of block 1.
- We will similarly do for all blocks.
- All of these queries have their opening in block 1, but their closing can be in any block including block 1.

MO's Algorithm

- Let's ignore all other blocks and focus on answering queries of block 1.
- We will similarly do for all blocks.
- All of these queries have their opening in block 1, but their closing can be in any block including block 1.
- Now, let us reorder these queries in ascending order of their R value. We do this for all the blocks.

Example

- Consider following queries and assume we have 3 blocks each of size 3.
- (0, 3) (1, 7) (2, 8) (7, 8) (4, 8) (4, 4) (1, 2)

Example

- Consider following queries and assume we have 3 blocks each of size 3.
- (0, 3) (1, 7) (2, 8) (7, 8) (4, 8) (4, 4) (1, 2)
- Let us re-order them based on their block number.

Example

- Consider following queries and assume we have 3 blocks each of size 3.
- (0, 3) (1, 7) (2, 8) (7, 8) (4, 8) (4, 4) (1, 2)
- Let us re-order them based on their block number.
- (0, 3) (1, 7) (2, 8) (1, 2) (4, 8) (4, 4) (7, 8)

Example

- Consider following queries and assume we have 3 blocks each of size 3.
- (0, 3) (1, 7) (2, 8) (7, 8) (4, 8) (4, 4) (1, 2)
- 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.

Example

- Consider following queries and assume we have 3 blocks each of size 3.
- (0, 3) (1, 7) (2, 8) (7, 8) (4, 8) (4, 4) (1, 2)
- 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)

Modified Solution

```
for each query:
    while L < currentL :
        currentL--
        add(currentL)
    while currentR < R:
        currentR++
        add(currentR)
    while currentL < L:
        remove(currentL)
        currentL++
    while R < currentR:
        remove(currentR)
        currentR--

    output answer
```

Proof of Time Complexity

Time Complexity of our algorithm is $O(N\sqrt{N})$.

Proof of Time Complexity

Time Complexity of our algorithm is $O(N\sqrt{N})$.

- The complexity over all queries is determined by the 4 while loops. Two while loops can be stated as "Amount moved by left pointer in total", other two while loops can be stated as "Amount moved by right pointer". Sum of these two will be the over all complexity.

Proof of Time Complexity

Time Complexity of our algorithm is $O(N\sqrt{N})$.

- The complexity over all queries is determined by the 4 while loops. Two while loops can be stated as "Amount moved by left pointer in total", other two while loops can be stated as "Amount moved by right pointer". Sum of these two will be the over all complexity.
- Let us talk about the right pointer first. For each block, the queries are sorted in increasing order, so clearly the right pointer `currentR` moves in increasing order. During the start of next block the pointer possibly at extreme end will move to least R in next block. That means for a given block, the amount moved by right pointer is $O(N)$. We have $O(\sqrt{N})$ blocks, so the total is $O(N\sqrt{N})$.

Proof of Time Complexity

- Let us see how the left pointer moves. For each block, the left pointer of all the queries fall in the same block, as we move from query to query the left pointer might move but as previous L and `currentL` fall in the same block, the moment is $O(\sqrt{N})$ (Size of the block). In each block the amount left pointer movies is $O(M\sqrt{N})$ where M is number of queries falling in that block. Total complexity is $O(M\sqrt{N}))$ for all blocks.

Proof of Time Complexity

- Let us see how the left pointer moves. For each block, the left pointer of all the queries fall in the same block, as we move from query to query the left pointer might move but as previous L and `currentL` fall in the same block, the moment is $O(\sqrt{N})$ (Size of the block). In each block the amount left pointer movies is $O(M\sqrt{N})$ where M is number of queries falling in that block. Total complexity is $O(M\sqrt{N}))$ for all blocks.
- Total complexity is $O((N + M)\sqrt{N}) = O(N\sqrt{N})$

MO's Algorithms on Trees

Subtree Queries

- **Problem:** Consider the following problem. You will be given a rooted Tree T of N nodes where each node is associated with a value $A[node]$. You need to handle Q queries, each comprising one integer u. In each query you must report the number of distinct values in the subtree rooted at u. In other words, if you store all the values in the subtree rooted at u in a set, what would be the size of this set?

Subtree Queries

- **Problem:** Consider the following problem. You will be given a rooted Tree T of N nodes where each node is associated with a value $A[node]$. You need to handle Q queries, each comprising one integer u. In each query you must report the number of distinct values in the subtree rooted at u. In other words, if you store all the values in the subtree rooted at u in a set, what would be the size of this set?
- **Solution:** Flatten the tree into an array by doing a Preorder traversal and then implement Mo's Algorithm.

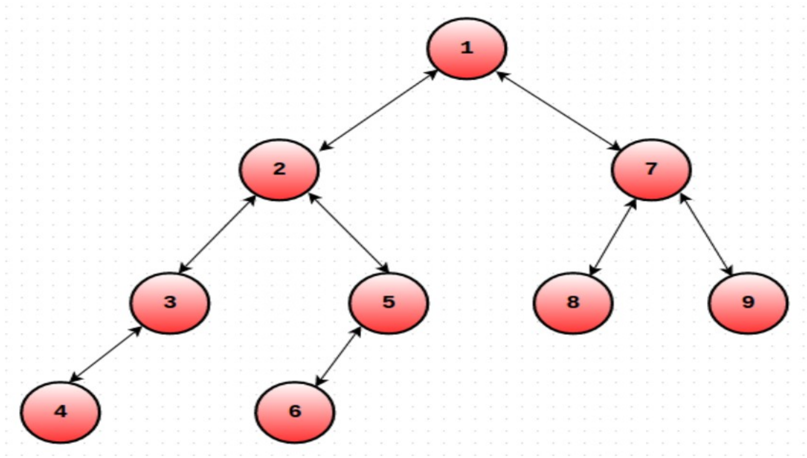
Path Queries

Problem: Instead of computing the number of distinct values in a subtree, compute the number of distinct values in the unique path from u to v.

Issues

Modified DFS-Order

Let us modify the dfs order as follows. For each node u, maintain the Start and End time of $S(u)$. Let's call them $ST(u)$ and $EN(u)$. The only change you need to make is that you must increment the global timekeeping variable even when you finish traversing some subtree ($EN(u) = ++cur$). In short, we will maintain 2 values for each node u. One will denote the time when you entered $S(u)$ and the other would denote the time when you exited $S(u)$.


1
2
7
3
5
8
9
4
6

$ST(1) = 1$ $EN(1) = 18$

$ST(2) = 2$ $EN(2) = 11$

$ST(3) = 3$ $EN(3) = 6$

$ST(4) = 4$ $EN(4) = 5$

$ST(5) = 7$ $EN(5) = 10$

$ST(6) = 8$ $EN(6) = 9$

$ST(7) = 12$ $EN(7) = 17$

$ST(8) = 13$ $EN(8) = 14$

$ST(9) = 15$ $EN(9) = 16$

$A[] = \{1, 2, 3, 4, 4, 3, 5, 6, 6, 5, 2, 7, 8, 8, 9, 9, 7, 1\}$

Algorithm

Let a query be (u, v). We will try to map each query to a range in the flattened array. Let $ST(u) \leq ST(v)$ where $ST(u)$ denotes visit time of node u in T. Let $P = LCA(u, v)$ denote the lowest common ancestor of nodes u and v. There are 2 possible cases:

Algorithm

Let a query be (u, v). We will try to map each query to a range in the flattened array. Let $ST(u) \leq ST(v)$ where $ST(u)$ denotes visit time of node u in T. Let $P = LCA(u, v)$ denote the lowest common ancestor of nodes u and v. There are 2 possible cases:

- Case 1: $P = u$,

Algorithm

Let a query be (u, v). We will try to map each query to a range in the flattened array. Let $ST(u) \leq ST(v)$ where $ST(u)$ denotes visit time of node u in T. Let $P = LCA(u, v)$ denote the lowest common ancestor of nodes u and v. There are 2 possible cases:

- Case 1: $P = u$,
- Our query range would be $[ST(u), ST(v)]$

Algorithm

Let a query be (u, v). We will try to map each query to a range in the flattened array. Let $ST(u) \leq ST(v)$ where $ST(u)$ denotes visit time of node u in T. Let $P = LCA(u, v)$ denote the lowest common ancestor of nodes u and v. There are 2 possible cases:

- Case 1: $P = u$,
- Our query range would be $[ST(u), ST(v)]$
- Case 2: $P \neq u$

Algorithm

Let a query be (u, v). We will try to map each query to a range in the flattened array. Let $ST(u) \leq ST(v)$ where $ST(u)$ denotes visit time of node u in T. Let $P = LCA(u, v)$ denote the lowest common ancestor of nodes u and v. There are 2 possible cases:

- Case 1: $P = u$,
- Our query range would be $[ST(u), ST(v)]$
- Case 2: $P \neq u$
- Our query range would be $[EN(u), ST(v)] + [ST(P), ST(P)]$.

Thank You