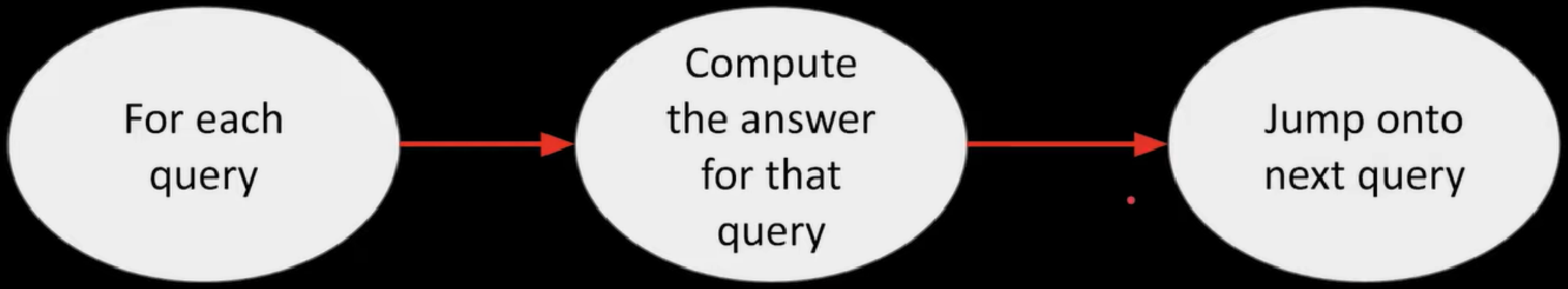
Mo's Algorithm

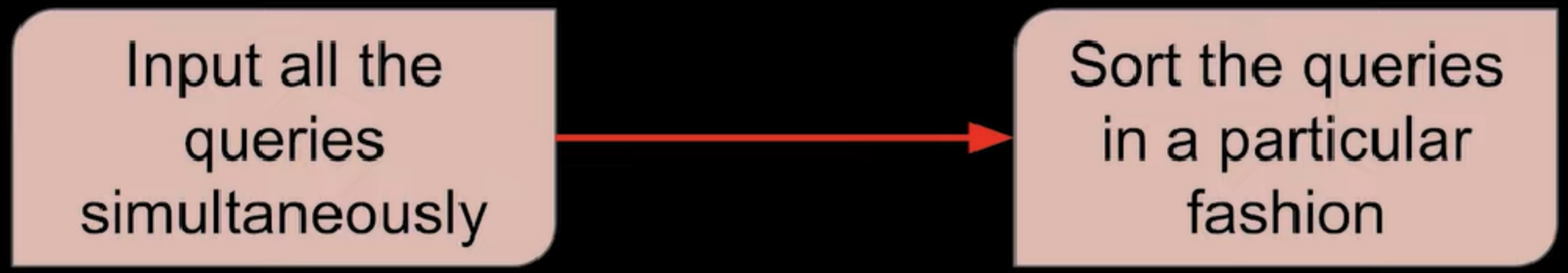
You are given an array of size n and q queries. Each query is a range query. In each query, we have to perform some operation and output answer of each query.

Brute force solution



Time Complexity = O(n*Q)

Efficient Solution - Mo's Algorithm



Note: This step is known as **offline processing** of queries

MO's Algorithm - Sorting Approach 1

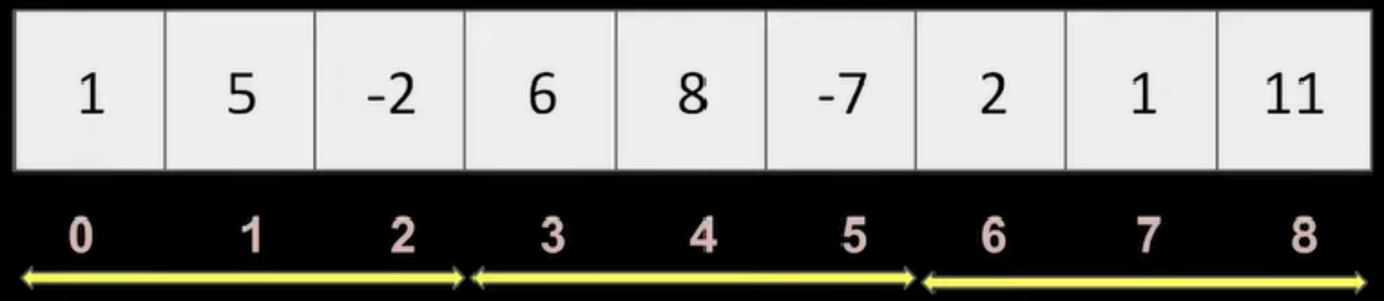
- Two queries with L in the same block are sorted as per increasing R.
- Two queries with L in different blocks are sorted as per increasing LB (L Block)

```
bool cmp(Query a, Query b) {
    return (a.lb < b.lb) ||
           (a.lb == b.lb && a.r < b.r);
}
```

MO's Algorithm - Sorting Approach 2

- We can (slightly) optimize the previous approach by sorting the R in reverse order for even blocks.

```
bool cmp(Query a, Query b) {
    return (a.lb < b.lb) ||
           (a.lb == b.lb &&
           (a.lb & 1 ? a.r < b.r : a.r > b.r));
}
```



Code:

```
int rootN;
bool compare(Q q1, Q q2) {
    if (q1.l / rootN == q2.l / rootN) {
        return q1.r > q2.r;
    }
    return q1.l / rootN < q2.l / rootN;
}
```

Example

l	r
7	8
1	6
2	7

l	r
2	7
1	6
7	8

Variables

curr_l curr_r

Pointers pointing to the current computed answer

l r

Pointers pointing to the required query.

Four cases

curr_l < l	curr_l > l	curr_r < r	curr_r > r
↓	↓	↓	↓
Increase curr_l	Decrease curr_l	Increase curr_r	Decrease curr_r

Dry Run

1	5	-2	6	8	-7	2	1	11
0	1	2	3	4	5	6	7	8

curr_l

l

r

curr_r

curr_sum = 25

Dry Run

1	5	-2	6	8	-7	2	1	11
0	1	2	3	4	5	6	7	8

curr_l l r curr_r

curr_sum = 24

Dry Run

1	5	-2	6	8	-7	2	1	11
0	1	2	3	4	5	6	7	8
		curr_l	l		r			curr_r

curr_sum = 19

Dry Run

1	5	-2	6	8	-7	2	1	11
0	1	2	3	4	5	6	7	8

curr_l l r curr_r

curr_sum = 21

Dry Run

1	5	-2	6	8	-7	2	1	11
0	1	2	3	4	5	6	7	8

curr_l l r curr_r

curr_sum = 9

Dry Run

1	5	-2	6	8	-7	2	1	11
0	1	2	3	4	5	6	7	8

curr_l l r curr_r

curr_sum = 7

Query's answer

MO's Algorithm - Sorting Approach 1

- We add/remove at most O(B) elements on the left side for every query -- O(B * Q).
- For every block, we add at-most O(N) elements on the right side -- O(N * N / B)
- For B = Sqrt(N), we get O((N + Q) * Sqrt(B)).

```
bool cmp(Query a, Query b) {
    return (a.lb < b.lb) ||
           (a.lb == b.lb && a.r < b.r);
}
```

Time Complexity

Sorting all queries → $Q\log(Q)$

Call for all blocks → $n/\sqrt{n} * n$

Instances of r

No. of blocks

No. of times change in value of curr_l → $\sqrt{n} * Q$

Total time complexity → $O((n + Q)\sqrt{n})$

MO's Algorithm - Sorting Approach 2

- We can (slightly) optimize the previous approach by sorting the R in reverse order for even blocks.

```
bool cmp(Query a, Query b) {
    return (a.lb < b.lb) ||
           (a.lb == b.lb &&
           (a.lb & 1 ? a.r < b.r : a.r > b.r));
}
```

```
struct query {
    int l, r, idx;
    query() { }
    query(int _l, int _r, int _i) : l(_l), r(_r), idx(_i) { }
    bool operator < (const query &p) const {
        if(l/block_sz != p.l/block_sz) return l < p.l;
        return ((l/block_sz) & 1) ? r > p.r : r < p.r;
    }
};

void mo(vector<query> &q) {
    sort(q.begin(), q.end());
    memset(ret, -1, sizeof ret);

    // l = 1, r = 0 if 1-indexed array
    int l = 0, r = -1;
    for(auto &qq : q) {
        while(qq.l < l) add(--l);
        while(qq.r > r) add(++r);
        while(qq.l > l) erase(l++);
        while(qq.r < r) erase(r--);
        ret[qq.idx] = max(ret[qq.idx], get_ans());
    }
}
```