Range Xor

We can solve this problem using trie data structure. At each node of the trie we will store the ith bit of the number for query of type 0. To solve the query of type 1 we will store the indexes of the number that pass through that node. When we go down the tree during a query and maximising the xor, we go in a direction that contains at least one index in range L to R, otherwise we go in the other direction. To search if atleast one index is within the stored indexes we will use binary search.

```java
import java.util.ArrayList;
import java.util.Scanner;

public class rangeXor {

    public static class trieNode {
        trieNode left;
        trieNode right;
        ArrayList<Integer> indexes;

        trieNode() {
            this.indexes = new ArrayList<Integer>();
        }
    }

    public static void insert(int n, trieNode head, int idx) {
        trieNode curr = head;
        for (int i = 31; i >= 0; i--) {
            int bit = (n >> i) & 1;
            if (bit == 0) {
                if (curr.left == null) {
                    curr.left = new trieNode();
                }
                curr.indexes.add(idx);
                curr = curr.left;
            } else {
                if (curr.right == null) {
                    curr.right = new trieNode();
                }
                curr.indexes.add(idx);
                curr = curr.right;
            }
        }
        curr.indexes.add(idx);
    }

    public static int maxXor(int value, trieNode head, int left, int right) {
        int curr_xor = 0;
        trieNode curr = head;
        for (int j = 31; j >= 0; j--) {
            int b = (value >> j) & 1;
```

```
            if (b == 0) {
                if (curr.right != null && binarySearchRange(curr.right.indexes,
left, right) ) {
                    curr = curr.right;
                    curr_xor += (int) Math.pow(2, j);
                } else {
                    curr = curr.left;
                }
            } else {
                if ( curr.left != null && binarySearchRange(curr.left.indexes,
left, right)) {
                    curr = curr.left;

                } else {
                    curr = curr.right;
                    curr_xor += (int) Math.pow(2, j);
                }
            }
        }
        return curr_xor;
    }

    public static boolean binarySearchRange(ArrayList<Integer> arr, int l, int
r) {
        int left = 0, right = arr.size() - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            int val = arr.get(mid);
            if(val>=l && val<=r) {
                return true;
            } else if (val <l) {
                left = mid+1;
            } else if (val >r) {
                right =mid-1;
            }
        }
        return false;
    }

    public static void main(String[] args) {

        Scanner scn = new Scanner(System.in);
        int q = scn.nextInt();
        trieNode head = new trieNode();
        int elementNumber = 0;
        for (int i = 0; i < q; i++) {
            int type = scn.nextInt();
            if (type == 0) {
                int val=scn.nextInt();
                insert(val, head, elementNumber++);
            } else if(type==1) {
                int l=scn.nextInt();
```

```
                int r=scn.nextInt();
                int x=scn.nextInt();
                System.out.println(maxXor(x, head, l-1, r-1));
            }
        }
    }

}
```