**Bit Manipulation**

1. Left shift x by n position -> x\*2n (works for small number).
2. Right shift x by n position ->x/2n (works for small number).
3. If(X&1) ==1 number is odd else even.
4. AND operation between multiple of 2 and next lower number always give 0. E.g. 8&7==0.
5. XOR operation between same number always give 0.E.g. 3^3=0.
6. X^0=X.
7. XOR helps in flip some specific bit. E.g.

5->0101

If we want to flip second bit, we make mask of ith bit and then,

0101^0100=0001;

1. For an integer num and you want to count the set bits of, repetitively apply a bitwise AND operation between num and (num-1).

By continuing this operation, one set bit at a time is cleared in num, and the count of bitwise AND operations performed becomes the count of set bits. num&(n-1)

Note: Set means 1 and Unset means 0

1. To find ith bit is Set or Unset:

If(num&(1<<i)==0) means bit is Unset.

Else: Set.

1. If we want to Set ith bit in a number, no matter ith bit is 0 or 1:

If ith bit is 0: [ num|(1<<i)];

1. If we want to Unset ith bit in a number, no matter ith bit is 0 or 1:

If ith bit is 1: [ num&(~(1<<i))];

1. Flipped last set bit.

Approach 1: num-(num&(-num).

Approach 2: num&(num-1).

Note: To calculate power of n of any number:

Formula: 4x =64 -> log(4x )=log(64) -> x log(4)=log(64) -> x=log(64)/log(4)

1. Number of 1’s bit in a number is using previous number :

Number of bits in (Number/2) + Number%2;(DP)

1. If X is the integer value of the first i binary digits, then adding the next binary digit A[i] is equivalent to 2X+A[i].

Let's illustrate this with an example:

* Binary array A=[1,0,1,1]

1. **First Digit**:
   * i=0
   * X=A[0]=1.
2. **Second Digit**:
   * i=1
   * X=2X+A[1]=2×1+0=2 .
   * Binary representation: 10
3. **Third Digit**:
   * i=2
   * X=2X+A[2]=2×2+1=5.
   * Binary representation: 101
4. **Fourth Digit**:
   * i=3
   * X=2X+A[3]=2×5+1=11.
   * Binary representation: 1011
5. XOR between two number helps to find number of different bits in number.

5^4 =1 (by counting number to bit in 1 we get the result)

**Q1.** **Convert decimal to binary.**

**public** **static** String dTc(**int** num) {

String str="";

**while**(num!=0) {

**int** temp=num%2;

str=temp+str;

num/=2;

}

**return** str;

}

**Q2. Convert binary to decimal.**

**public void** bTd(int num) {

**int** n=1100;

**int** i=0;

**int** num=0;

**while** (n! =0) {

**int** digit =n%10;

**if**(digit==1) {

num=num+(**int**)Math.*pow*(2, i);

}

n=n/10;

i++;

}

System.***out***.println(num);

**Q3**. **Find complement of a number.**

 public int findComplement(int num) {

    int temp=0;

    while(temp<num) {

        temp=(temp<<1)|1;

    }

    return temp-num;

    }}

**Q4**. **XOR of a given number range:**

If n=4 then , ans=1^2^3^4=4;

**Code**: Brute Force. O(n)

    public int m1(int n) {

         int result=0;

      for(int i=1;i<=n;i++) {

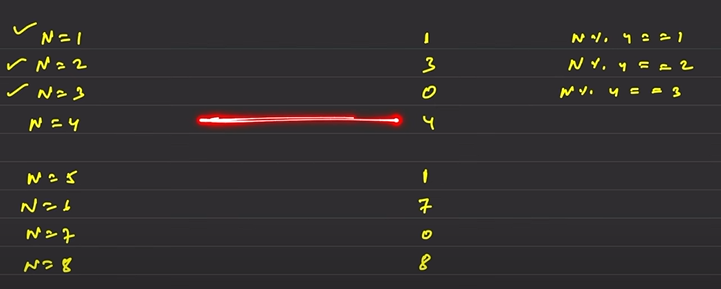
          result=result^i

              }

     return count;

      }

**Code**: Optimized. O(1)



There is a pattern if we check above. The pattern is repeat after every 4 number.

**public** **static** **int** m1(**int** n) {

**if**(n%4==0){

**return** n;

}

**else** **if** (n%4==1) {

**return** 1;

}

**else** **if** (n%4==2) {

**return** n+1;

}

**return** 0;

}

If we want to find XOR from range (n , m): e.g.(4,7):(1^2^3)^(1^2^3^4^5^6^7)

We can use above function:

**public** **static** **int** m1(**int** n ,m) {

**return** m1(n-1)^m1(m);

**}**

**Q5**. **Counting’s Bits**

Given an integer n, return an array ans of length n + 1 such that for each i (0 <= i <= n), ans[i] is the ***number of***1***'s*** in the binary representation of i.

#Bit Manipulation #DP

When n=5; num= [0,1,2,3,4,5] Output-> arr= [0,1,1,2,1,2]

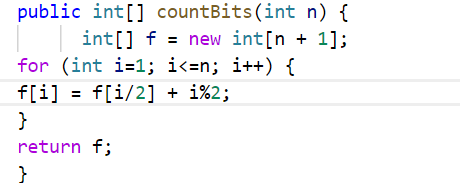
Case1: num[i], when i is even/odd then number of bit at i-indexed i/2+i%2;

E.g. num[4] =num[4/2]+i%2; =num[2]+0 = 2;

num[5] =num[5/2]+i%2; =num[2]+1 = 3;

If i is even, then i%2 is always Zero else 1.

**Code:**

****

**Q5**. **Reverse Bit’s.**

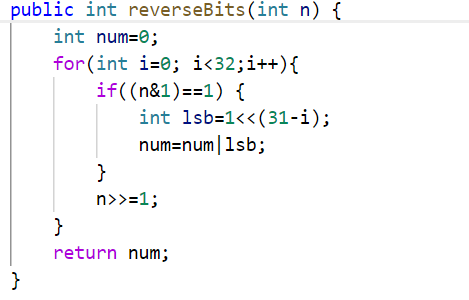
**Approach**: First, we have to find least significant bit ; (a integer consist of 32-bit)

E.g. 00000000000000000000010101010101 , take a bit mask (say 1) and left shift it by position say (31) , 1000000000000000000000000000000 and take number(let num=0)

and do OR operation i.e. num=bitmask|num=1000000000000000000000000000000;

Makes first bit at its exact position. Repeat it 32 times.

**Code:**

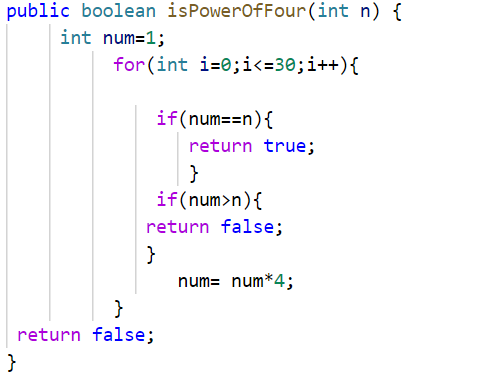
****

**Q6.** **Power of 4.**

Range of integer is 231-1 =230 =( (2)2)15  = 415 By running loop 30 times we check any number

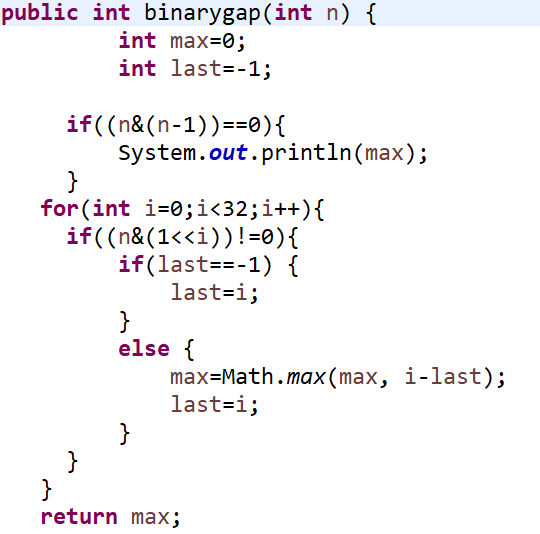
by replacing 4;

**Code:**

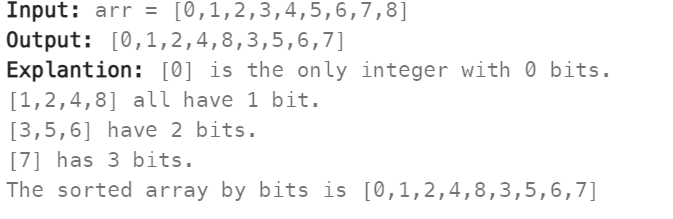
****

**Q7. Binary Gap (maximum distance between two adjacent 1 having no 1 between them.)**

**Code:**  we have to track the all gap between 1 and return the maximum.

****

**Q8. Sort a given array** based **on the number of bits;(IMP).**

****

Brute Force Solution:

|  |  |
| --- | --- |
| Bit Count | Value |
| 0 | [0] |
| 1  2  3 | [1,2,3,4]  [3,5,6]  [7] |

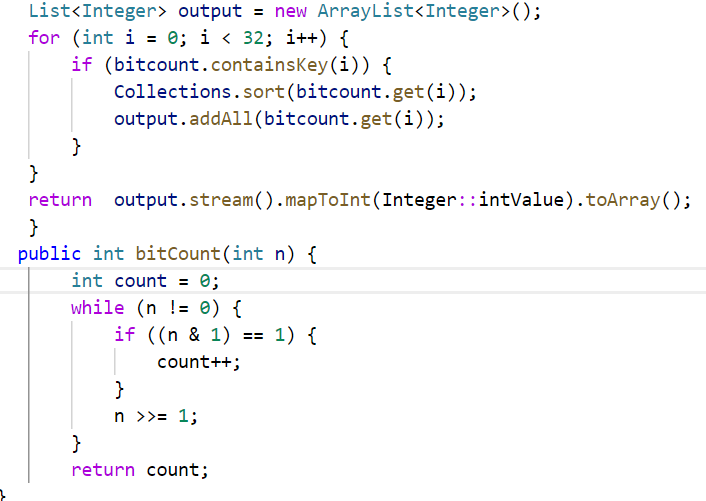
First add all elements have 0 bit in a list ,then 1 bit in a list …………………………….;

Sort individual list and then add all list in one list and convert to array.

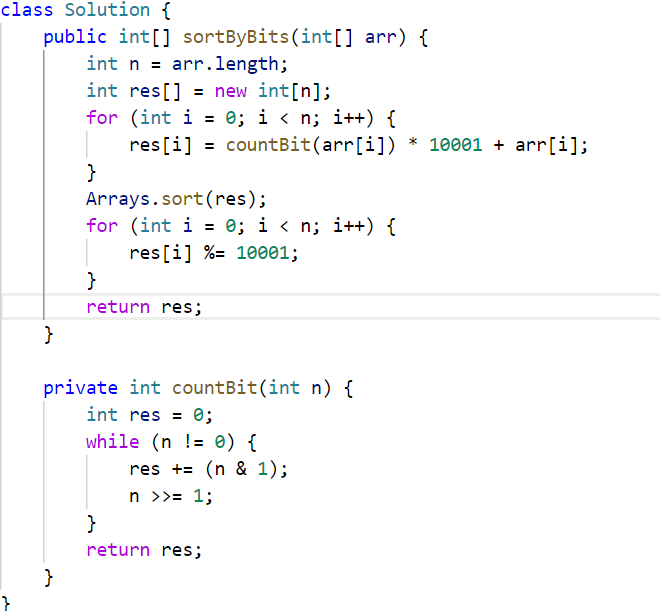
Note: 2 and 4 have same bit then in final array 2 must come first. No matter what the order in input.

**Code:**

****

****

Another approach:



**Q9: Count Pairs of Similar Strings;( Two strings are similar if they consist of the same characters.)**

* For example, "abca" and "cba" are similar since both consist of characters 'a', 'b', and 'c'.
* However, "abacba" and "bcfd" are not similar since they do not consist of the same characters.
* **Input:** words = ["aba","aabb","abcd","bac","aabc"]
* **Output:** 2

First Approach:(Using hashSet)

class Solution {

    public int similarPairs(String[] words) {

         int count=0;

      for(int i=0;i<words.length-1;i++) {

          for(int j=i+1;j<words.length;j++) {

              if(checkSimilar(words[i], words[j])) {

                  count++;

              }

          }

      }

      return count;

    }

    static boolean checkSimilar(String s1,String s2) {

        Set<Character> s1set= new HashSet<Character>();

        Set<Character> s2set= new HashSet<Character>();

        for(char c:s1.toCharArray()) {

            s1set.add(c);

        }

        for(char c:s2.toCharArray()) {

            s2set.add(c);

        }

        if(s1set.equals(s2set)) {

            return true;

        }

        return false;

    }

}

Approach 2:(Using Bit Manipulation):

OR operation between same number always give same number.(3 | 3 = 3)

class Solution {

static int bitMask(String s) {

        int mask=0;

        for(char c:s.toCharArray()) {

            mask|=(1<<(c-'a'));

    }

        return mask;

    }

     public int similarPairs(String[] words) {

        int count=0;

      int[] bitCount=new int[words.length];

      for(int i=0;i<words.length;i++) {

        bitCount[i]=bitMask(words[i]);

          }

      for(int i=0;i<words.length-1;i++) {

          for(int j=i+1;j<words.length;j++) {

          if(bitCount[i]==bitCount[j]) {

              count++;

          }

      }

    }

    return count;

    }

}

**Q10. Single Number ll.(LC-137)**

Every element appears **three times** except for one, which appears **exactly once**. *Find the single element and return it*.

Approach 1:(Bit Manipulation).

public int singleNumber(int[] nums) {

int num=0;

for(int i=0;i<32;i++) {

int count=0;

for(int j=0;j<nums.length;j++) {

if((nums[j]&(1<<i))!=0) {

count++;

}

}

if(count%3==1) {

num=num|(1<<i);

}

}

return num;

}

Approach 2:(Using sorting)

     static int v1() {

        int num=0;

        int[] arr = {0,11,0,11,0,11,9};

        Arrays.sort(arr);

        for(int i=1;i<arr.length;i+=3) {

            if(arr[i]!=arr[i-1]) {

                return arr[i-1];

            }

        }

        return arr[arr.length-1];

    }

**Q11. Single Number lll.(LC-260)**

Given an integer array nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once. You can return the answer in **any order**.

class Solution {

public int[] singleNumber(int[] nums) {

int[] arr= new int[2];

int num=0;

for(int i=0;i<nums.length;i++){

num=num^nums[i];

}

int bitnum=num^(num&(num-1));//to count first set bit(it give bit mask )

int b1=0,b2=0;

for(int i=0;i<nums.length;i++){

if((nums[i]&bitnum)!=0){

b1=b1^nums[i];

}

else{

b2=b2^nums[i];

}

}

arr[0]=b1;

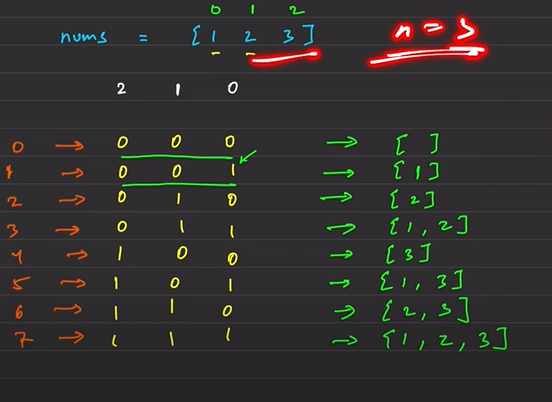
arr[1]=b2;

return arr;

}

**Q13. Subsets of an array when all elements are distinct.(using bit manipulation).**

No of subset of array having length n is : 2n or(1<<n).



Code:

 public List<List<Integer>> subsets(int[] nums) {

       List<List<Integer>> list=new LinkedList<List<Integer>>();

        int subset=1<<(nums.length);

        for(int i=0;i<subset;i++) {

            List<Integer> sublist= new ArrayList<Integer>();

            int j=0;

            while(j<32) {

                if((i&(1<<j))!=0) {

                    sublist.add(nums[j]);

                }

                j++;

            }

            list.add(sublist);

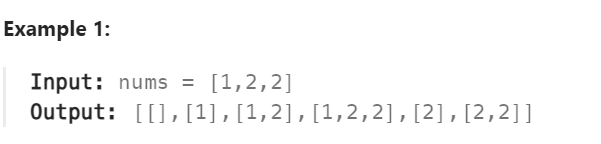
        }

        return list;

    }

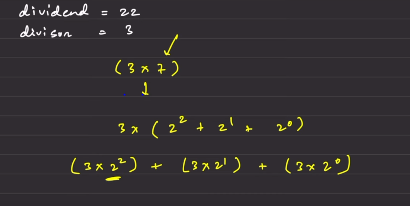
**Q14. Subsets II.(LC-90).(IMP).**

Given an integer array nums that may contain duplicates, return all possible subsets(power set).The solution set must not contain duplicate subsets. Return in any order.



**Q15. Divide two number without using divide and multiplication sign.(IMP)**

Code:



**static** **int** divide (**int** dividend, **int** divisor) {

**boolean** sign=**true**;

**if** (dividend==divisor)

**return** 1;

**if** (divisor<0 && dividend>0) {

sign=**false**;

}

**if** (dividend<0 && divisor>0) {

sign=**false**;

}

**long** n = Math.*abs*((**long**) divisor);

**long** divid =Math.*abs*((**long**) dividend);

**long** ans=0;

**while** (divid >= n) {

**int** count = 0;

**while** (divid >=(n<<count + 1)) {

count++;

}

ans+=(1<<count);

divid =divid-(n << count);

}

**if** ((ans==(1<<31)) && sign) {

**return** Integer.***MAX\_VALUE***;

}

**if** ((ans ==(1 << 31)) && !sign) {

**return** Integer.***MIN\_VALUE***;

}

**return** (**int**) (sign?ans:-(ans));

}

**Q18. Sum of all subsets XOR totals.(LC-1863)**

The XOR total of an array is defined as the bitwise XOR of all its subsets, or 0 if the array is empty.

**First Approach**: Find all subsets XOR and then add up all.

Code:

  public int subsetXORSum(int[] nums) {

        int sum=0;

        int subset= 1<<nums.length;

        for(int i=0;i<subset;i++){

            int j=0;

            int sumXOR=0;

            while(j<32){

                if((i&(1<<j))!=0){

                    sumXOR=sumXOR^nums[j];

                }

                j++;

            }

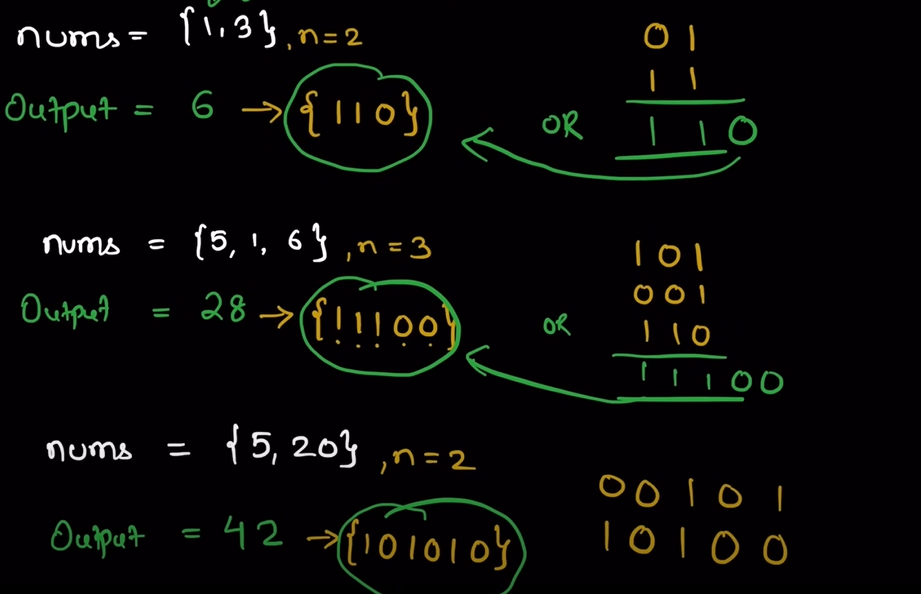
            sum=sum+sumXOR;

        }

        return sum;

    }

**Approach 2**: If we observe the pattern is OR of all number and then left shift by n-1;



CODE:

public int subsetXORSum(int[] nums) {

      int length = nums.length;

        int xorBits = 0;

        for (int i =0;i<length;i++){

            xorBits |= nums[i];

        }

        int sum = (int) (xorBits\* (1<<length-1));

        return sum;

    }

**Q19. Bitwise AND of number Range.(LC-210)**

Given two numbers **left** and **right** that represent range [left, right] return *the bitwise AND of all numbers in this range, inclusive*.

Approach 1: Start with right = right & (right-1) up to right> left; In every step rightmost set bit becomes zero and store result of & operation in result. Maximum time complexity is O(32).

Code:

public int rangeBitwiseAnd(int left, int right) {

while(right>left) {

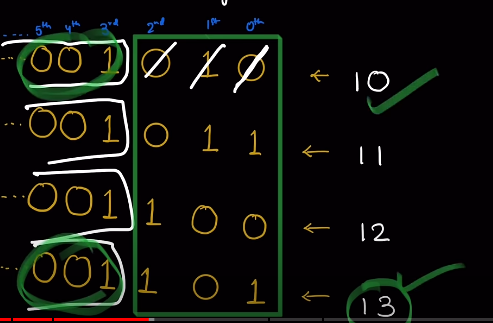
right=right&(right-1);

}

return right;

}

Approach 2: Right shift right and left until both become equal and count the no of operation. And then the number comes left shift number by count.

Code:

public int rangeBitwiseAnd(int left, int right) {

int count=0;

while (left!=right) {

left>>=1;

right>>=1;

count++;

}

return (left<<count);

**Q20. Missing number in an array. (LC-268).**

Given an array nums containing n distinct numbers in the range [ 0, n], return *the only number in the range that is missing from the array.*

*Code:*

public int missingNumber(int[] nums) {  
 int n=nums.length;   
 int ans=0;  
 for(int i=0; i<nums.length;i++) {  
 ans=ans^nums[i]^i;

return ans^n;

}

**Q21. Find duplicate in an array. (LC-287).**

Make negative every visited index element and when a negative come which means to revisited the same element, return the element.

public int findDuplicate(int[] nums) {

for (int i=0 ; i<nums.length;i++) {

int x=Math.abs(nums[i]) ;

if(nums[x]<0) {

return x ;

}

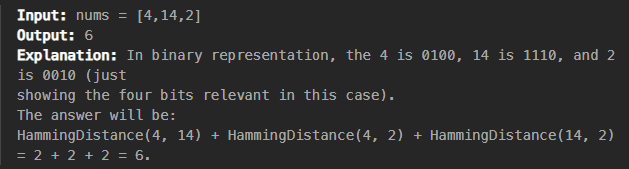
nums [x]=-nums[x] ;

}

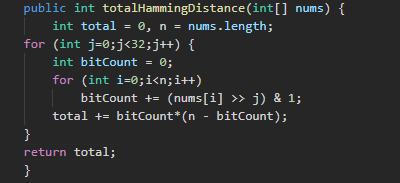
return -1 ;

**Q22. Total hamming distance. (LC-477).**

The hamming distance between two integer is number of positions at which corresponding bits are different.

Approach:

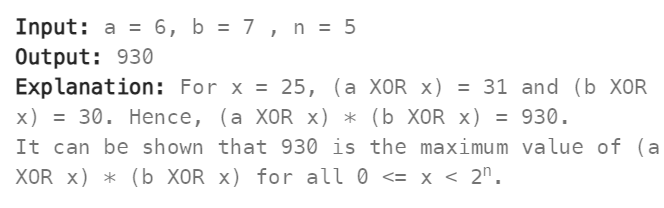
For each bit position 1-32 in a 32-bit integer, we count the number of integers in the array which have that bit set. Then, if there are n integers in the array and k of them have a particular bit set and (n-k) do not, then that bit contributes k\*(n-k) hamming distance to the total.



**Q23. Find Maximum XOR Product. (LC-2939).**

Give two integer a ,b and n , return the maximum value of (a XOR x) \* (b XOR x) where 0<=x<2n. Since answer may be too large, return it modulo 109+7;

0<=a ,b <250. 0<=n<=50; Instead of calculating x we direct calculate a^x .



Code:

  public int maximumXorProduct(long a, long b, int n) {

        long aXOR = 0;

        long bXOR = 0;

        for(long i = 49;i>=n;i--)

        {

            boolean a\_ith\_bit = ((a >> i) & 1)> 0;

            boolean b\_ith\_bit = ((b >> i) & 1) > 0;

            if(a\_ith\_bit==true){

                aXOR ^=(1L<<i);

            }

            if(b\_ith\_bit==true){

                bXOR ^=(1L<<i);

            }

        }

        for(long i = n-1;i>=0;i--){

            boolean a\_ith\_bit = ((a >> i) & 1)> 0;

            boolean b\_ith\_bit = ((b >> i) & 1) > 0;

            if(a\_ith\_bit == b\_ith\_bit){

                aXOR^=(1L<<i);

                bXOR^=(1L<<i);

            }

           else {

                    if (aXOR > bXOR) {

                        bXOR ^= (1L << i);

                    } else {

                        aXOR ^= (1L << i);

                    }

                }

            }

          aXOR%=1000000007;

          bXOR%=1000000007;

          return

}