Number Theory

Content

Introduction

In this newsletter, we will be looking at some applications of the concepts learned in the previous newsletter. We will be covering some questions and their approach in this newsletter.

Applications

1. **Check if a number is odd or even**

   Approach

   The idea is to check whether the last bit of the number is set or not. If the last bit is set then the number is odd, otherwise even.

   As we know bitwise XOR Operation of the Number by 1 increments the value of the number by 1 if the number is even, otherwise it decrements the value of the number by 1 if the value is odd.

   Implementation

Visit the link -
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Number%20Theory/Codes/Bitwise/oddOrEven.cpp

2. **Given an array of integers. All numbers occur twice except one number which occurs once. Find the number in O(n) time & constant extra space.**

   Approach 1

   One solution is to check every element if it appears once or not. Once an element with a single occurrence is found, return it. Time complexity of this solution is O(n2).

   Approach 2

   A better solution is to use hashing.

   1) Traverse all elements and put them in a hash table. Element is used as key and the count of occurrences is used as the value in the hash table.

   2) Traverse the array again and print the element with count 1 in the hash table.

   This solution works in O(n) time but requires extra space.

   Approach 3

   The best solution is to use XOR. XOR of all array elements gives us the number with a single occurrence. The idea is based on the following two facts.

   a) XOR of a number with itself is 0.

   b) XOR of a number with 0 is the number itself.

   Implementation

   Visit the link -
   https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Number%20Theory/Codes/Bitwise/findUniqueInDuplicatesArray.cpp

3. **Find ith bit of a number.**

   Approach

   1) Find a number with all 0s except k-th position. We get this number using (1 << (k-1)). For example if k = 3, then (1 << 2) gives us (00..00100).

2) Do bitwise and above-obtained numbers with n to find if k-th bit in n is set or not.

Implementation

Visit the link - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Number%20Theory/Codes/Bitwise/findIthBitOfANumber.cpp

4. **Set the ith bit of a number**

   Approach

   To set any bit we use bitwise OR | operator. As we already know bitwise OR | operator evaluates each bit of the result to 1 if any of the operand's corresponding bit is set (1). In-order to set kth bit of a number we need to shift 1 k times to its left and then perform bitwise OR operation with the number and result of left shift performed just before.

   Implementation

   Visit the link - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Number%20Theory/Codes/Bitwise/setIthBitOfANumber.cpp

5. **Reset the ith bit of a number**.

   Approach - Since bitwise AND of any bit with a reset bit results in a reset bit, i.e.Any bit <bitwise AND> Reset bit = Reset bit which means, 0 & 0 = 0 and 1 & 0 = 0. So for clearing a bit, performing a bitwise AND of the number with a reset bit is the best idea. n = n & ~(1 << k) OR n &= ~(1 << k) where k is the bit that is to be cleared.

6. **Find position of rightmost set bit**

   Approach

   Let I/P be 12 (1100)

   1. Take two's complement of the given no as all bits are reverted except the first '1' from right to left (0100).
   2. Do a bit-wise & with original no, this will return no with the required one only (0100).

3. Take the log2 of the no, you will get (position – 1) (2).
4. Add 1 (3).

Implementation

Visit the link - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Number%20Theory/Codes/Bitwise/findPositionOfRightMostSetBit.cpp

7. **Given an array where every element occurs three times, except one element which occurs only once. Find the element that occurs once.**

Approach

We can use sorting to do it in O(nLogn) time. We can also use hashing, it has the worst-case time complexity of O(n), but requires extra space.

The idea is to use bitwise operators for a solution that is O(n) time and uses O(1) extra space. The solution is not easy like other XOR-based solutions, because all elements appear an odd number of times here. The idea is taken from here.

Run a loop for all elements in the array. At the end of every iteration, maintain the following two values.

ones: The bits that have appeared 1st time or 4th time or 7th time .. etc.

twos: The bits that have appeared 2nd time or 5th time or 8th time .. etc.

Finally, we return the value of 'ones'

How to maintain the values of 'ones' and 'twos'?

'ones' and 'twos' are initialized as 0. For every new element in the array, find out the common set bits in the new element and the previous value of 'ones'. These common set bits are actually the bits that should be added to 'twos'. So do bitwise XOR of the common set bits with 'twos'. 'twos' also gets some extra bits that appear the third time. These extra bits are removed later.

Update 'ones' by doing XOR of new element with the previous value of 'ones'. There may be some bits that appear 3rd time. These extra bits are also removed later.

Both 'ones' and 'twos' contain those extra bits which appear 3rd time. Remove these extra bits by finding out common set bits in 'ones' and 'twos'.

Implementation

Visit the link -
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Number%20Theory/Codes/Bitwise/findUniqueAmongArrayOfTriplets.cpp

8. **Find nth magic number.**

   A magic number is defined as a number which can be expressed as a power of 5 or sum of unique powers of 5. First few magic numbers are 5, 25, 30(5 + 25), 125, 130(125 + 5), ….

   Approach

   If we notice carefully the magic numbers can be represented as 001, 010, 011, 100, 101, 110 etc, where 001 is 0*pow(5,3) + 0*pow(5,2) + 1*pow(5,1). So basically we need to add powers of 5 for each bit set in given integer n.

   Implementation

   Visit the link -
   https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Number%20Theory/Codes/Bitwise/findNthMagicNumber.cpp

9. **Given a number, find if its a power of 2 or not.**

   Approach

   A simple method for this is to simply take the log of the number on base 2 and if you get an integer then the number is the power of.

   Implementation

   Visit the link -
   https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Number%20Theory/Codes/Bitwise/checkIfPowerOf2.cpp

10. **Find no. of set bits in a given number.**

    Approach

    Brian Kernighan's Algorithm:

    Subtracting 1 from a decimal number flips all the bits after the rightmost set bit(which is 1) including the rightmost set bit.

for example :

10 in binary is 00001010

9 in binary is 00001001

8 in binary is 00001000

7 in binary is 00000111

So if we subtract a number by 1 and do it bitwise & with itself (n & (n-1)), we unset the rightmost set bit. If we do n & (n-1) in a loop and count the number of times the loop executes, we get the set bit count.

The beauty of this solution is the number of times it loops is equal to the number of set bits in a given integer.

Implementation

Visit the link - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Number%20Theory/Codes/Bitwise/countNoOfSetBits.cpp

11. **Find xor of 0 or 1 to a number a.**

    Approach

    Method 1 (Naive Approach):

    1- Initialize result as 0.

    1- Traverse all numbers from 1 to n.

    2- Do XOR of numbers one by one with result.

    3- At the end, return result.

    Method 2 (Efficient method) :

    1- Find the remainder of n by moduling it with 4.

    2- If rem = 0, then xor will be same as n.

    3- If rem = 1, then xor will be 1.

    4- If rem = 2, then xor will be n+1.

    5- If rem = 3 ,then xor will be 0.

    Implementation

Visit the link -
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Number%20Theory/Codes/Bitwise/xorFromZeroToA.cpp

12. **Find xor of all numbers between 2 ranges [L, R] (including both L and R).**

Approach

Naive Approach: Initialize answer as zero, Traverse all numbers from L to R and perform XOR of the numbers one by one with the answer. This would take O(N) time.

Efficient Approach: By following the approach discussed here, we can find the XOR of elements from the range [1, N] in O(1) time.

Using this approach, we have to find xor of elements from the range [1, L – 1] and from the range [1, R] and then xor the respective answers again to get the xor of the elements from the range [L, R]. This is because every element from the range [1, L – 1] will get XORed twice in the result resulting in a 0 which when XORed with the elements of the range [L, R] will give the result.

Implementation

Visit the link -
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Number%20Theory/Codes/Bitwise/rangeXOR.cpp