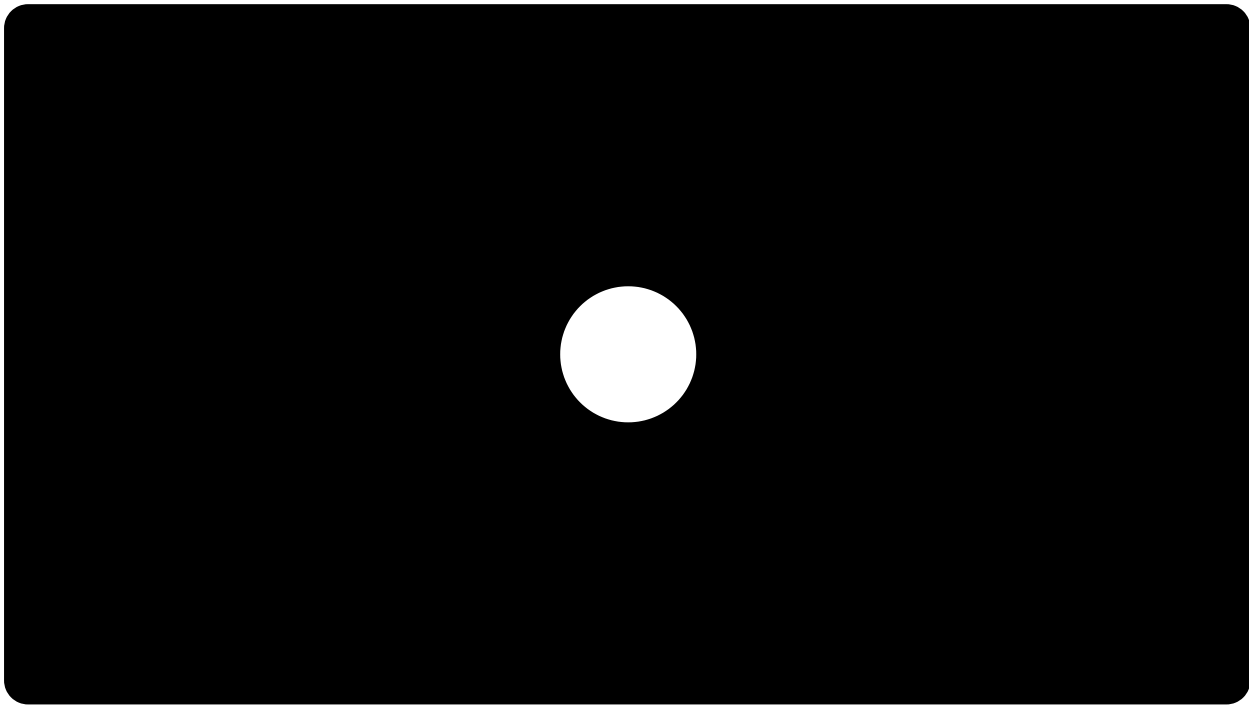
Contribution technique on bits



Problem Statement: You are given array a containing n elements. You have to find ΣΣ(a[i]^a[j]).

Code:

Copy

```
int n, int arr[n], int ans=0;

for(int j=0; j<31; j++)
{
    int cnt_0=0, cnt_1=0;

    for(int i=0; i<n; i++)
    {
        if(arr[i]&(1<<j))
        {
            cnt_1++;
        }
        else
        {
            cnt_0++;
        }
```

```
    }

    int pairs = cnt_0 * cnt_1;

    ans += pairs * (1<<j);
}
cout << ans;
```

Explanation:

N :	1	3	5	
	0	0	1	---> 2 * 2^2
	0	1	0	---> 2 * 2^2
	1	1	1	---> 0 * 2^0

An pair can give 1 in XOR operation if it contain 0 and 1.

1. **Outer Loop (`for(int j=0; j<31; j++)`):**
    - This loop iterates over each bit position from 0 to 30 (assuming 32-bit integers, as the loop runs up to j<31).
2. **Inner Loop (`for(int i=0; i<n; i++)`):**
    - This loop iterates over each element of the array.
    - Inside the loop, the code checks the `j`-th bit of each array element (`arr[i] & (1<<j)`).
    - If the `j`-th bit is set (i.e., it's 1), it increments `cnt_1`; otherwise, it increments `cnt_0`.
3. **Counting Pairs:**
    - After the inner loop, the code calculates the number of pairs with different bit values at position `j` using `cnt_0` and `cnt_1`.
    - `pairs = cnt_0 * cnt_1`.
4. **Updating the Answer (`ans`):**
    - The code updates the final answer by adding the contribution of the pairs at the current bit position.
    - `ans += pairs * (1<<j)`.
5. **Print the Final Result:**
    - Finally, the code prints the accumulated sum of XOR pairs.

It's important to note that `(1<<j)` is a bitwise left shift operation, which effectively represents 2^j. This is used to calculate the contribution of the pairs at the `j`-th bit position.

In summary, the algorithm iterates over each bit position, counts the number of 0s and 1s at that position, calculates the number of pairs with different bit values, and accumulates the sum accordingly. The algorithm takes advantage of bitwise operations to efficiently process each bit position.

Time Complexity:

1. **Outer Loop:**
    - The outer loop runs for a constant number of iterations (31 in this case), as it iterates over each bit position.
    - Time complexity: O(1)
2. **Inner Loop:**
    - The inner loop iterates over each element in the array, performing constant-time operations for each iteration.
    - The bitwise operations inside the inner loop (`arr[i] & (1<<j)`) take constant time.
    - Time complexity: O(n)

Since the outer loop is constant time and the inner loop is O(n), the overall time complexity of the algorithm is O(1) * O(n), which simplifies to O(n).

Now, lets see a property of bit manipulation which will be used in problem solving.

`a+b=a|b+a&b`

Let's break down how this expression is derived:

1. **a|b:** This part represents the bitwise OR operation between a and b. In binary addition, if there is no carry, the sum of corresponding bits is obtained by taking the bitwise OR of the bits. For example, if a=1010 and b=0101, then a|b=1111 because for each bit position, we take the OR of the corresponding bits (1 OR 0 is 1).

2. **a&b:** This part represents the bitwise AND operation between a and b. In binary addition, if there is a carry, it is equivalent to performing a bitwise AND of the bits. The result of this AND operation gives the positions where a carry occurs. For example, if a=1101 and b=0101, then a&b=0101 because for each bit position, we take the AND of the corresponding bits (1 AND 0 is 0).

3. **Putting it together:** When there is no carry, a+b=a|b as explained in the first part. However, when there is a carry, we add the carry back to the sum. The carry is