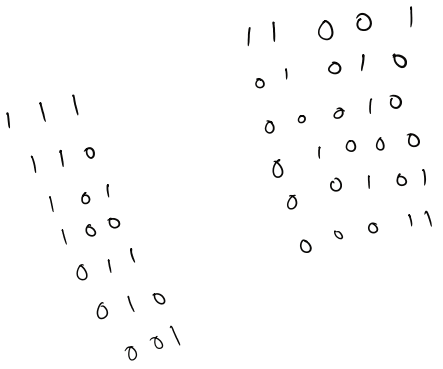
Saturday, April 19, 2025 6:53 PM

Maximum XOR of two numbers in an array

eg: I/p: arr[] = [25, 10, 2, 8, 5, 3]

O/p: 28

[11001, 1010, 10, 1000, 101, 11]



Maximum XOR of two numbers in an array

Difficulty: **Medium** Accuracy: **50.0%** Submissions: **16K+** Points: **4**

Given an array **arr[]** of non-negative integers of size **n**. Find the maximum possible **XOR** between two numbers present in the array.

Examples:

Input: arr[] = [25, 10, 2, 8, 5, 3]
Output: 28
Explanation: The maximum possible XOR is 5 ^ 25 = 28.

Input: arr[] = [1, 2, 3, 4, 5, 6, 7]
Output: 7
Explanation : The maximum possible XOR is 1 ^ 6 = 7.

Constraints:
$2 \leq arr.size() \leq 5*10^4$
$1 \leq arr[i] \leq 10^6$

[Naive Approach] Using 2 Nested Loops - O(n^2) Time and O(1) Space

The idea is to **generate all possible pairs** of elements of the array arr[] and find the maximum among them. To do so, use two nested loops to generate all the pairs and compute XOR of them. The maximum of all the pairs is the result.

C++ Java Python C# JavaScript

```cpp
#include <bits/stdc++.h>
using namespace std;

// Function to find the maximum XOR
int maxXor(vector<int> &arr) {
    int res = 0;
    int fir = 0, sec = 0;
    // Generate all possible pairs
    for (int i = 0; i < arr.size(); i++) {
        for (int j = i + 1; j < arr.size(); j++) {
            if((arr[i]^arr[j]) > res){
              res = max(res, arr[i] ^ arr[j]);
             fir = arr[i], sec = arr[j];
            }
            }
    }
    return res;
}

int main() {
    vector<int> arr = {26, 100, 25, 13, 4, 14};
    cout << maxXor(arr);
    return 0;
}
```

Output

126

[Expected Approach - 1] - Using Bit-Masking & Hashset

- $O(n * \log m)$ Time and $O(n * \log m)$ space

The idea is to find two numbers in an array arr[] such that their XOR equals a number X, where X is the maximum value we want to achieve at the current bit position (i-th bit).

To achieve the largest XOR value, we aim to maximize the number of 1's in the XOR result, starting from MSB to LSB.

*To evaluate each bit position, we use a mask. A mask helps us focus on specific bits of the numbers in the array by keeping only the relevant bits up to the current position and ignoring the rest. Using the mask, we extract **prefixes** for all numbers in the array (i.e., the portions of the numbers defined by the mask). These prefixes help us determine if a pair of numbers exists in the array whose XOR can yield a maximum value at the current bit.*
*For **each** bit position:*

- *Apply the mask to extract prefixes from the numbers.*
- *Try to update the maximum XOR value by assuming the i-th bit of the result is 1.*
- *Using the set of prefixes, we check if any two prefixes can produce the desired XOR value (with the i-th bit set).*

This process is repeated for all 32 bits, starting from the leftmost bit, to compute the largest possible XOR value step by step.

```cpp
class Solution {
  public:
    int maxXor(vector<int> &arr) {
        // code here
       int res=0, mask=0;

        // to store all unique bits
        unordered_set<int> s;

       for(int i=30;i>=0;i--){

            // set the i-th bit in mask
            mask |=(1<<i);

            for(auto value : arr){

                // keep prefix of all elements till the i-th bit
                s.insert(value & mask);
            }
            int cur = res | (1<<i);
            for(int prefix : s){
                if(s.count(cur^prefix)){
                    res=cur;
                    break;
                }
            }
            s.clear();
        }
       return res;
    }
};
```

[Expected Approach - 2] Using Trie – O(n * (log m)) Time & O(n * log m) Space :-

The idea is to use Trie data structure to effectively store & search bits of each element of array arr[i]., check if an element with opposite bits is present in the Trie or not i.e. if current bit of arr[i] is set (1), then check if unset bit (0) at current index is present in Trie.

```
// } Driver Code Ends

// User function Template for C++
class Node{
    public:
    Node* one;
    Node* zero;
    Node(){
        one = nullptr;
        zero = nullptr;
    }
};
class Trie{
    public:
    Node* root;
    Trie(){
        root = new Node();
    }
    // Function to insert in Trie
    void insert(int n){
        Node* curr = root;
        for(int i=31;i>=0;i--){
            int bit = (n>>i) & 1;
            // Check if the bit is 0
            if(bit == 0){
                if(curr -> zero == NULL){
                    curr->zero = new Node();
                }
                curr = curr->zero;
            }

            // Else if bit is 1
            else{
                if(curr->one == NULL){
                    curr->one = new Node();
                }
                curr = curr->one;
            }
        }
    }
    // Function to find element having
    // the maximum XOR value with n
    int findXOR(int n){
        Node* curr = root;
        int res = 0;
        for(int i=31;i>=0;i--){
            int bit=(n>>i)&1;
            // If the bit is 0
            if(bit==0){
                // if set bit is present
                if(curr->one){
                    curr=curr->one;
                    res+=(1<<i);
                }
                else{
                    curr=curr->zero;
                }
            }
            // Else if bit is 1
            else{
                // if unset is present
                if(curr -> zero){
                    curr = curr->zero;
                    res+=(1<<i);
                }
                else{
                    curr=curr->one;
                }
            }
        }
        return res;
    }
};
class Solution {
  public:
    int maxXor(vector<int> &arr) {
        // code here
        int res=0;
        Trie* t=new Trie();

        // Insert the first element in trie
        t->insert(arr[0]);
        for(int i=1;i<arr.size();i++){
            res=max(res, t->findXOR(arr[i]));
            t->insert(arr[i]);
        }
        return res;
    }
};
```