# Question

Given an integer array nums, return *the maximum result of nums[i] XOR nums[j]*, where 0 ≤ i ≤ j < n.

**Follow up:** Could you do this in O(n) runtime?

**Example 1:**

**Input:** nums = [3,10,5,25,2,8]

**Output:** 28

**Explanation:** The maximum result is 5 XOR 25 = 28.

**Example 2:**

**Input:** nums = [0]

**Output:** 0

**Example 3:**

**Input:** nums = [2,4]

**Output:** 6

**Example 4:**

**Input:** nums = [8,10,2]

**Output:** 10

**Example 5:**

**Input:** nums = [14,70,53,83,49,91,36,80,92,51,66,70]

**Output:** 127

**Constraints:**

* 1 <= nums.length <= 2 \* 104
* 0 <= nums[i] <= 231 - 1

# Solution

#### **Overview**

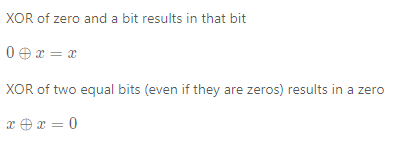
Requirements are to have \mathcal{O}(N)O(*N*) time complexity, and we'll discuss here two standard approaches to achieve that complexity.

1. Bitwise Prefixes in HashSet.
2. Bitwise Prefixes in Trie.

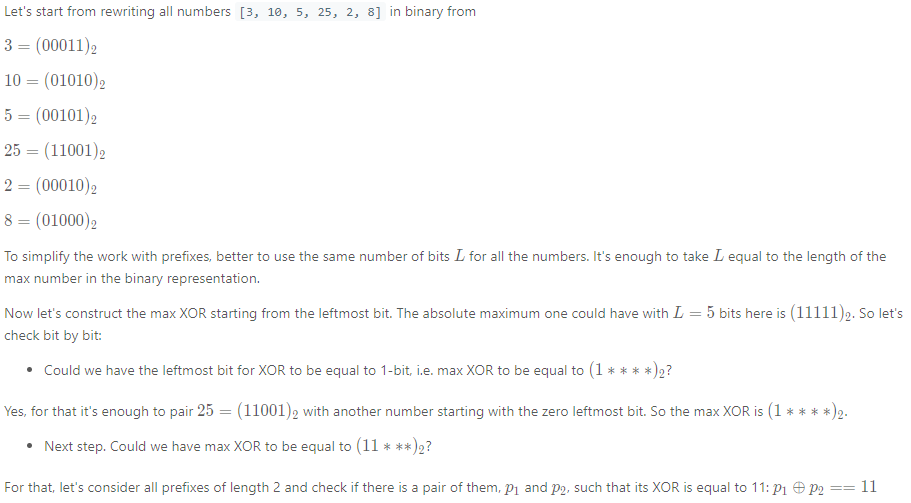
The idea behind both solutions is the same: to convert all numbers into the binary form, and to construct the maximum XOR bit by bit, starting from the leftmost one. The difference is in the data structure used to store unique bitwise prefixes, i.e. the first ith bits.

The first approach works faster on the given testcase set, but the second one is standard, more simple, and easily generalised for more complex problems like Find maximum subarray XOR in a given array.

**Prerequisites**



#### **Approach 1: Bitwise Prefixes in HashSet**



3=(00∗∗∗)2​

10=(01∗∗∗)2​

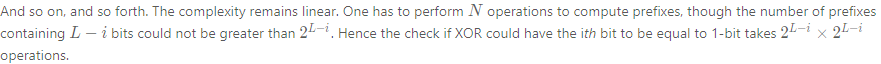
5=(00∗∗∗)2​

25=(11∗∗∗)2​

2=(00∗∗∗)2​

8=(01∗∗∗)2​

Yes, it's the case, for example, pair 5=(00∗∗∗)2​ and 25=(11∗∗∗)2​, or 2=(00∗∗∗)2​ and 25=(11∗∗∗)2​, or 3=(00∗∗∗)2​ and 25=(11∗∗∗)2​.

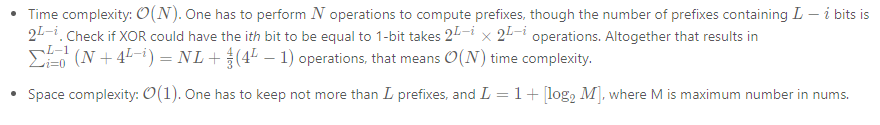


**Algorithm**

* Compute the number of bits L*L* to be used. It's a length of max number in binary representation.
* Initiate max\_xor = 0.
* Loop from i = L - 1*i*=*L*−1 down to i = 0*i*=0 (from the leftmost bit L - 1*L*−1 to the rightmost bit 0):
  + Left shift the max\_xor to free the next bit.
  + Initiate variable curr\_xor = max\_xor | 1 by setting 1 in the rightmost bit of max\_xor. Now let's check if curr\_xor could be done using available prefixes.
  + Compute all possible prefixes of length L - i*L*−*i* by iterating over nums.
    - Put in the hashset prefixes the prefix of the current number of the length L - i*L*−*i*: num >> i.
  + Iterate over all prefixes and check if curr\_xor could be done using two of them: p1^p2 == curr\_xor. Using self-inverse property of XOR p1^p2^p2 = p1, one could rewrite it as p1 == curr\_xor^p2 and simply check for each p if curr\_xor^p is in prefixes. If so, set max\_xor to be equal to curr\_xor, i.e. set 1-bit in the rightmost bit. Otherwise, let max\_xor keep 0-bit in the rightmost bit.
* Return max\_xor.

|  |
| --- |
| class Solution {  public int findMaximumXOR(int[] nums) {  int maxNum = nums[0];  for(int num : nums) maxNum = Math.max(maxNum, num);  // length of max number in a binary representation  int L = (Integer.toBinaryString(maxNum)).length();  int maxXor = 0, currXor;  Set<Integer> prefixes = new HashSet<>();  for(int i = L - 1; i > -1; --i) {  // go to the next bit by the left shift  maxXor <<= 1;  // set 1 in the smallest bit  currXor = maxXor | 1;  prefixes.clear();  // compute all possible prefixes  // of length (L - i) in binary representation  for(int num: nums) prefixes.add(num >> i);  // Update maxXor, if two of these prefixes could result in currXor.  // Check if p1^p2 == currXor, i.e. p1 == currXor^p2.  for(int p: prefixes) {  if (prefixes.contains(currXor^p)) {  maxXor = currXor;  break;  }  }  }  return maxXor;  }  } |

**Complexity Analysis**



#### **Approach 2: Bitwise Trie**

**Why HashSet is not a Good Structure to Store Prefixes**

Hashset structure, used to store the prefixes in Approach 1, doesn't provide the functionality to cut off some paths which don't lead to the solution.

For example, after two steps of max XOR computation (11\*\*\*)\_2(11∗∗∗)2​ it's quite obvious that 25 should be paired with 0000 prefix, i.e. with 2, 3, or 5.

3 = (00011)\_23=(00011)2​

10 = (01010)\_210=(01010)2​

5 = (00101)\_25=(00101)2​

25 = (11001)\_225=(11001)2​

2 = (00010)\_22=(00010)2​

8 = (01000)\_28=(01000)2​

Although for the third step we'll again compute all possible prefixes, including the ones for 10 and 8, even if it's quite obvious that they will not lead to the solution.

3 = (000\*\*)\_23=(000∗∗)2​

10 = (010\*\*)\_210=(010∗∗)2​

5 = (001\*\*)\_25=(001∗∗)2​

25 = (110\*\*)\_225=(110∗∗)2​

2 = (000\*\*)\_22=(000∗∗)2​

8 = (010\*\*)\_28=(010∗∗)2​

To cut these branches off, would be great to use some sort of tree structure.

**Bitwise Trie: What is it and How to Construct**

The standard way is to use [Bitwise Trie](https://en.wikipedia.org/wiki/Trie#Bitwise_tries). It's a special type of [Trie](https://leetcode.com/articles/word-search-ii/), which is used to store binary prefixes in an efficient way. There are plenty of real-life examples of bitwise trie usage, for example, [in GCC](https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/trie_based_containers.html).

Let's start with Bitwise Trie for the array [3, 10, 5, 25, 2]

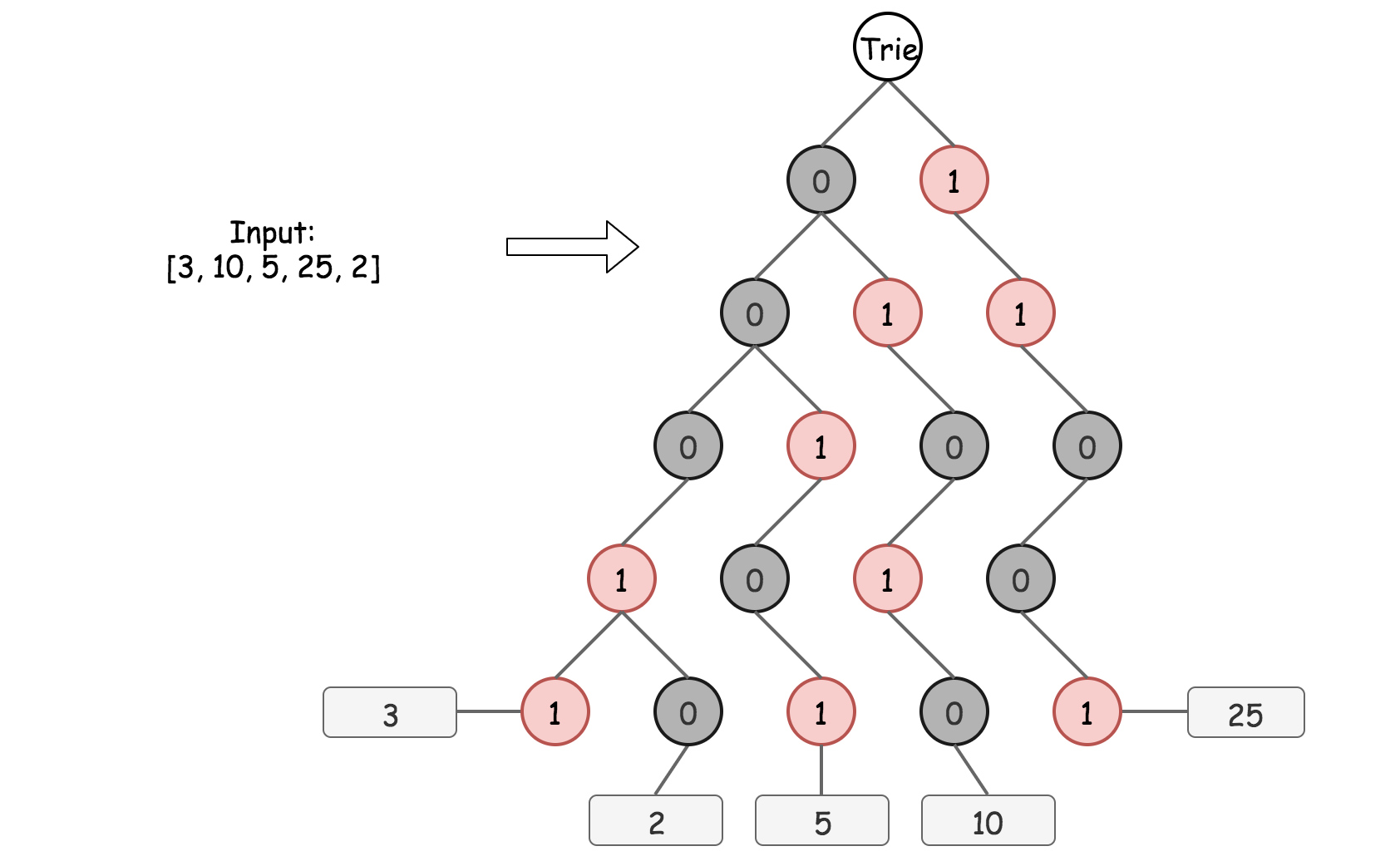
3 = (00011)\_23=(00011)2​

10 = (01010)\_210=(01010)2​

5 = (00101)\_25=(00101)2​

25 = (11001)\_225=(11001)2​

2 = (00010)\_22=(00010)2​



Each root -> leaf path in Bitwise Trie represents a binary form of a number in nums, for example, 0 -> 0 -> 0 -> 1 -> 1 is 3. As before, the same number of bits L*L* is used for all numbers, and L = 1 + [\log\_2 M]*L*=1+[log2​*M*], where M is a maximum number in nums. The depth of Bitwise Trie is equal to L*L* as well, and all leafs are on the same level.

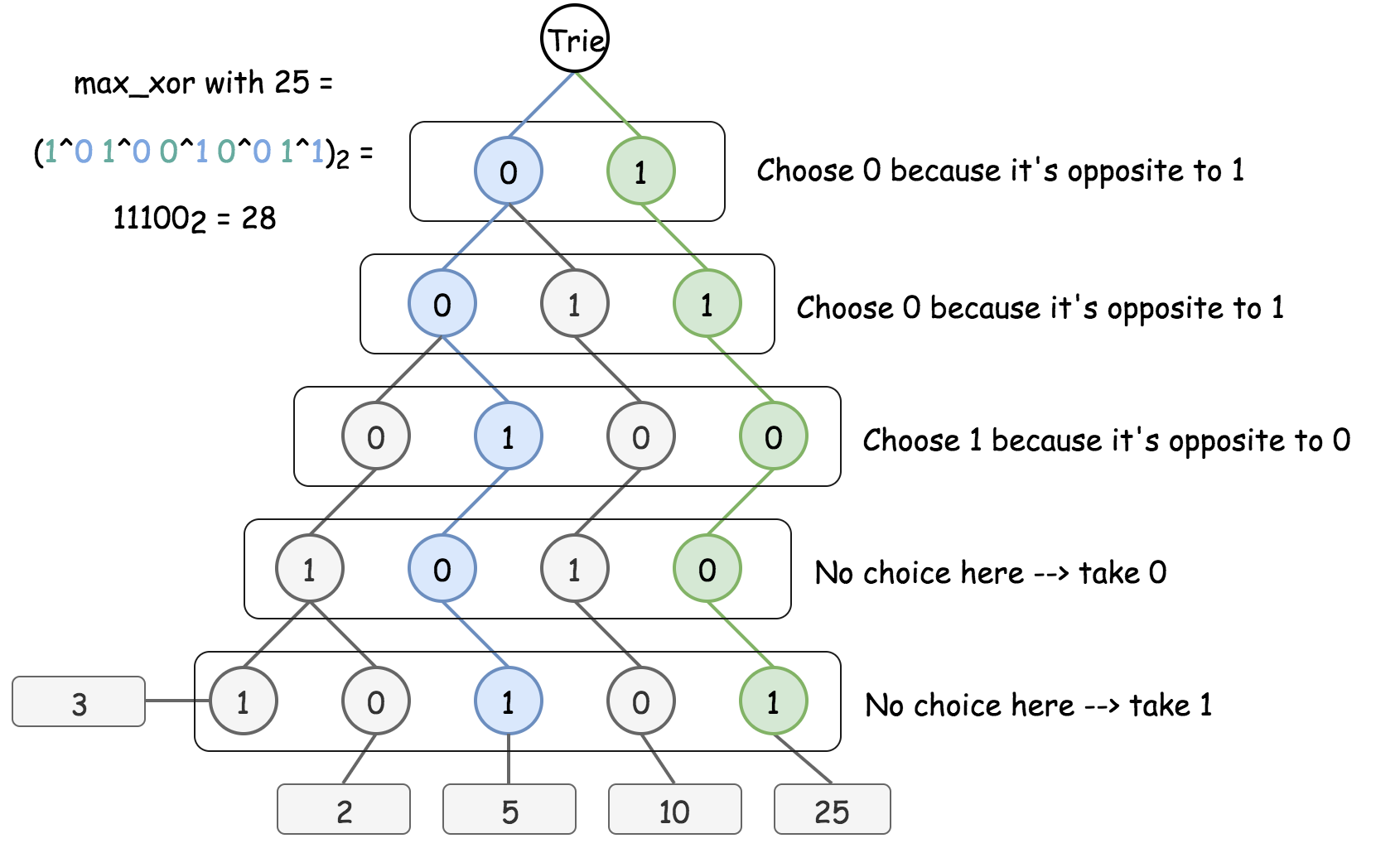
Bitwise Trie is a perfect way to see how different the binary forms of numbers are, for example, 3 and 2 share 4 bits of 5. The construction of Bitwise Trie is pretty straightforward, it's basically nested hashmaps. At each step one has to verify, if the child node to add (0 or 1) is already present. If yes, just go one step down. If not, add it into the Trie and then go one step down.

|  |
| --- |
| TrieNode trie = new TrieNode();  for (String num : strNums) {  TrieNode node = trie;  for (Character bit : num.toCharArray()) {  if (node.children.containsKey(bit)) {  node = node.children.get(bit);  } else {  TrieNode newNode = new TrieNode();  node.children.put(bit, newNode);  node = newNode;  }  }  } |

**Maximum XOR of a Given Number with All Numbers in Trie**

Now the Trie is constructed, so let's find the maximum XOR of a given number with all numbers that have been already inserted into Bitwise Trie.

To maximize XOR, the strategy is to choose the opposite bit at each step whenever it's possible. Step by step for 25 as a given number:



The implementation is also pretty simple:

* Try to go down to the opposite bit at each step if it's possible. Add 1-bit at the end of current XOR.
* If not, just go down to the same bit. Add 0-bit at the end of current XOR.

|  |
| --- |
| TrieNode trie = new TrieNode();  for (String num : strNums) {  TrieNode xorNode = trie;  int currXor = 0;  for (Character bit : num.toCharArray()) {  Character toggledBit = bit == '1' ? '0' : '1';  if (xorNode.children.containsKey(toggledBit)) {  currXor = (currXor << 1) | 1;  xorNode = xorNode.children.get(toggledBit);  } else {  currXor = currXor << 1;  xorNode = xorNode.children.get(bit);  }  }  } |

**Algorithm**

To summarise, now one could

* Insert a number into Bitwise Trie.
* Find maximum XOR of a given number with all numbers that have been inserted so far.

That's all one needs to solve the initial problem:

* Convert all numbers to the binary form.
* Add the numbers into Trie one by one and compute the maximum XOR of a number to add with all previously inserted. Update maximum XOR at each step.
* Return max\_xor.

**Implementation**

|  |
| --- |
| class TrieNode {  HashMap<Character, TrieNode> children = new HashMap<Character, TrieNode>();  public TrieNode() {}  }  class Solution {  public int findMaximumXOR(int[] nums) {  // Compute length L of max number in a binary representation  int maxNum = nums[0];  for(int num : nums) maxNum = Math.max(maxNum, num);  int L = (Integer.toBinaryString(maxNum)).length();  // zero left-padding to ensure L bits for each number  int n = nums.length, bitmask = 1 << L;  String [] strNums = new String[n];  for(int i = 0; i < n; ++i) {  strNums[i] = Integer.toBinaryString(bitmask | nums[i]).substring(1);  }  TrieNode trie = new TrieNode();  int maxXor = 0;  for (String num : strNums) {  TrieNode node = trie, xorNode = trie;  int currXor = 0;  for (Character bit : num.toCharArray()) {  // insert new number in trie  if (node.children.containsKey(bit)) {  node = node.children.get(bit);  } else {  TrieNode newNode = new TrieNode();  node.children.put(bit, newNode);  node = newNode;  }  // compute max xor of that new number  // with all previously inserted  Character toggledBit = bit == '1' ? '0' : '1';  if (xorNode.children.containsKey(toggledBit)) {  currXor = (currXor << 1) | 1;  xorNode = xorNode.children.get(toggledBit);  } else {  currXor = currXor << 1;  xorNode = xorNode.children.get(bit);  }  }  maxXor = Math.max(maxXor, currXor);  }  return maxXor;  }  } |

**Complexity Analysis**

