Todays Content:

- Check Bit Revise ✓
- Unique Element I ✓
- Unique Element II
- Unique Element III ✓
    a) Updated version of Unique element-III ✓
- Sum of all xor pairs

Problems:

$0 <= N <= 10^9$
$0 <= i <= 30$
} For given N check if i^{th} bit is set or Not?
1 0

```
bool CheckBit(N, i){
    return [(N>>i)&1==1]
}
```

	6	5	4	3	2	1	0
N=53 :	0	1	1	0	1	0	1

i=0 :

N&1==1 : 0^{th} bit set

i=1

	6	5	4	3	2	1	0
N=53 :	0	1	1	0	1	0	1
N>>1 :		0	1	1	0	1	0

(N>>1)&1==1 : 1^{st} bit set

i=2

	6	5	4	3	2	1	0
N=53 :	0	1	1	0	1	0	1
N>>2 :			0	1	1	0	1

(N>>2)&1==1 : 2^{nd} bit set

Unique element:

Every element repeats twice except 1, find unique element?

Ex1: ar[7] = { 3 2 3 7 2 8 7 } : ans = 8

Ex2: ar[9] = { 3 6 2 3 5 4 5 6 4 } : ans = 2

Idea: xor of all elements:

TC: O(N) SC: O(1)

Unique element II

Given ar[N], every element repeats thrice except 1, find the unique element?
↳ comes 1 times

Constraints:

$1 <= N <= 10^6$

$1 <= ar[i] <= 10^9$

ar[7] = 6 5 6 4 5 6 5 → ans = 4

ar[13] = 5 7 5 4 7 11 11 9 11 7 5 4 4 → ans = 9

a) for every element, iterate in array get freq == 1

TC: N * {O(N)} ⇒ O(N²) SC: O(1)

optimize: Optimize using hashmap, to get freq of elements

TC: O(N + N) = O(N) SC: O(N)

b) Sort the arr[], & iterate on arr[] & get unique elements

TC: O(NlogN + N) ⇒ O(NlogN) SC: O(1)

c) xor of all elements:

ar[7] = 6 ^ 5 ^ 6 ^ 4 ^ 5 ^ 6 ^ 5 = 6 ^ 5 ^ 4 ≠ final ans

ar[13] = 5 ^ 7 ^ 5 ^ 4 ^ 7 ^ 11 ^ 11 ^ 9 ^ 11 ^ 7 ^ 5 ^ 4 ^ 4 =

= 4 ^ 9 ^ 11 ^ 7 ^ 5

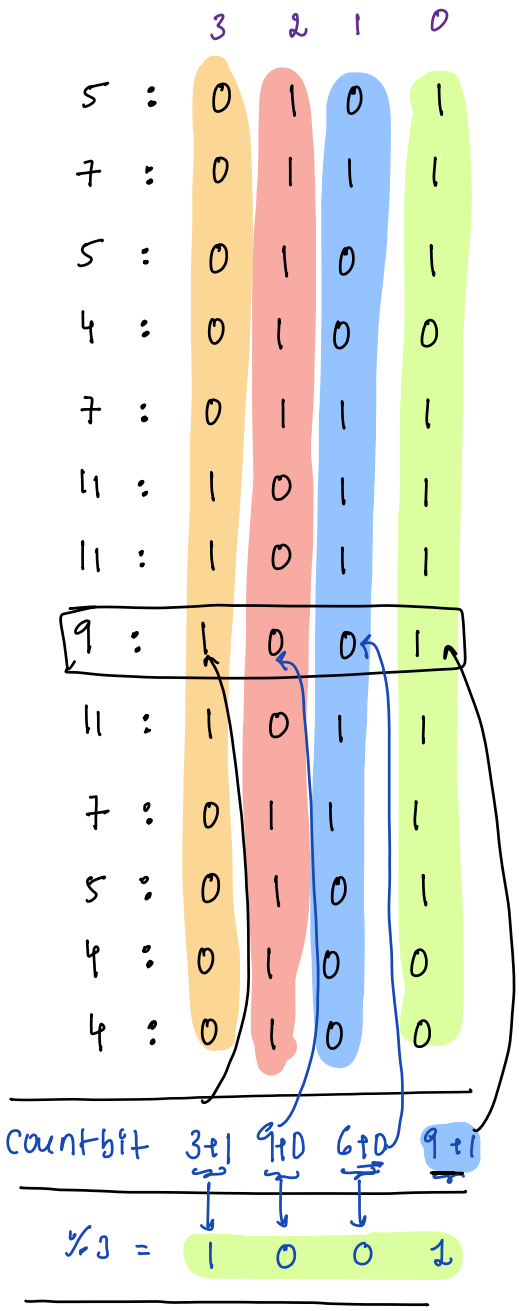
Ideas:

In general any Bit manipulation based question try to solve bit by bit?

Contribution technique works
↳ based on bit by bit pos?

Idea3:

ar[13] = 5 7 5 4 7 11 11 9 11 7 5 4 4



: Idea: at every bit calculate no: of set bits :

Say at i^{th} bit no: of arr[] elements with i^{th} bit set = c_i

if(c_i % 3 != 0) => In unique element i^{th} bit set

Pseudo Code:

```
int TripleTrouble(int ar[], int n){        TC: 32*N → O(N)
                                           SC: O(1)
    ans = 0;
    i = 0; i<32; i++){
        // for ith bit, calculate no: of ar[] with ith bit set ?
        ci = 0
        j = 0; j<n; j++){
            if(checkbit(ar[j], i) == True){        1<<i = 2^i
                ci = ci + 1
            }
        }
        if (ci % 3 != 0){ // for unique ele ith bit is set ?
            ans = ans + (1<<i) // 2^i, because decimal for ith bit = 2^i
        }
    }
    return ans;
}
```

Extention:

Every element repeats thrice except 1 element repeats 2 time

→ Every element repeats 4 times

→ Except 1 element, repeat 1 time

: xor of all elements

→ Except 1 element, repeat 2 time

: xor of all elements : { a^a^a^b^a^b^c^c^c^c }

: if (ci % 4 != 0){ // for unique ele ith bit is set }

→ Except 1 element, repeat 3 time :

: xor of all elements : { a^a^a^b^a^b^c^c^c^c^b }

Unique Element - [1]

Given ar[N], every element repeats twice except 2 elements which occur 1 time

find the 2 unique elements

Eg:

ar[6] = { 3 6 4 4 3 8 } : ans = {6, 8}

ar[4] = { 4 9 9 8 } : ans = { 4, 8 }

ar[6] = { 2 4 4 6 6 2 } not possible

Ideas:

a) for every element, iterate in array get freq == 1

TC: N * {O(N)} ⇒ O(N²) SC: O(1)

optimize: Optimize using hashmap, to get freq of elements

TC: O(N + N) = O(N) SC: O(N)

b) Sort the arr[], iterate on arr[] & get unique elements

ar[8]: { 6 4 6 3 7 3 9 7 }

Sort : { 3 3 4 6 6 7 7 9 } : ans = {4, 9}

TC: O(NlogN + N) ⇒ O(NlogN) SC: O(1)

c) XOR of all elements

ar[6] = { 3 ^ 6 ^ 4 ^ 4 ^ 3 ^ 8 } = 6 ^ 8 = 14

ar[4] = { 4 ^ 9 ^ 9 ^ 8 } = 4 ^ 8 = 12

Obs: xor of all ele = {xor of unique elements}

	1010	1000	1100	0110	1010	1100
ar[12] =	10	8 8	9	12	9	6 11	10	6	12	17
	1000	1001	1001	1011	0110	10001

Step 1: xor of all ar[]

v = 11 ^ 17

	4	3	2	1	0
17 :	1	0	0	0	1
11 :	0	1	0	1	1
v =	1	1	0	1	0

Obs: for v 1st bit is 1

Both unique have diff diff value at bit pos = 1

Obs: for v 3rd bit is 1

Both unique have diff diff value at bit pos = 3

Obs: for v 4th bit is 1

Both unique have diff diff value at bit pos = 4

Iterate on arr[], split it based on 1st bit data

- Set: all elements with 1st bit set = 1 → [10 6 11 10 6] → xor of all ele: 11
- Unset: all elements with 1st bit unset = 0 → [8 8 9 12 9 12 17] → xor of all ele: 17

Iterate on arr[], split it based on 3rd bit data

- Set: all elements with 3rd bit set = 1 → [10 8 8 9 12 9 11 10 12] → xor of all ele = 11
- Unset: all elements with 3rd bit unset = 0 → [6 6 17] → xor of all ele = 17

Pseudocode:

TC: O(N+32+N) ≈ O(N)
SC: O(1)

```
int UniqueII(int arr[], int n){

    Step1: xor of all ele
        int v=0;
        i=0; i<n; i++) { v= v^arr[i] }
        // v = xor of 2 unique elements

    Step2: find bit pos based on which we need to split
        p = __                    { Take any set bit, that's fine }
        i=0; i<32; i++){
            if(checkBit(v, i) == True){
                // Both unique elements different data at ith bit pos
                p = i; break
            }
        }

    Step3: Split arr[] based on bit pos = p
        set = 0, unset = 0
        i=0; i<n; i++){
            if(checkbit(arr[i], p) == True){
                // arr[i] goes to set space
                set = set^arr[i]
            }
            else{
                // arr[i] goes to unset space
                unset = unset^arr[i]
            }
        }
    // 2 unique elements are set & unset
}
```

3Q)

Given ar[N], contains all elements from [1, N+2] except 2 ele

find 2 missing elements?

ar[4] = {3 6 1 4} : missing ele = 2 5 (range {1-6})

ar[5] = {1 6 4 7 5} : missing ele = 2 3 (range {1-7})

ar[4] = {3 3 4 2} : missing ele = 1, 5, 6 (range {1-6})

Ideas:
a) 2 nested loops
b) Using hashmap
c) Sort
d) Bringing ele to it's corct pos → Doubt Session
e) Create mathematical equations

XOR idea:

ar[5] = {1^6^4^7^5} {1^2^3^4^5^6^7} (range {1, 7})

Idea: every element repeats twice, except 2, find 2 unique elements

are 2 missing elements

→ TC: O(N) SC: O(1) : TODO

→ Will discuss in todays doubt Session

Pseudocode : TODO/ Doubts

→ maths equations :

arr[N] :→ in arr[] missing elements are a & b

↳ {1, N+2}

All elements from [1, N+2] — all arr[] elements

{1 2 3 .. a .. b .. N+2} {1 2 3 .... N+2}

eq1 : Sum of all [1, N+2] − Sum of arr[] elem = a+b

eq2 : Sum of square [1, N+2] − sum of square arr[] ele = a^2+b^2

Say : $a+b = V_1$ ✓

$a^2+b^2 = V_2$ ✓

↳ square = $(a+b)^2 = V_1^2 = a^2+b^2+2ab = V_1^2$

$= V_2 + 2ab = V_1^2$

$(a-b)^2 = a^2+b^2-2ab$ $\quad \boxed{2ab = V_1^2 - V_2}$

$= V_2 - (V_1^2 - V_2)$

$= V_2 - V_1^2 + V_2$

$(a-b)^2 = 2V_2 - V_1^2$

$a-b = \sqrt{2V_2 - V_1^2}$

$a+b = V_1$

Calculate a & b

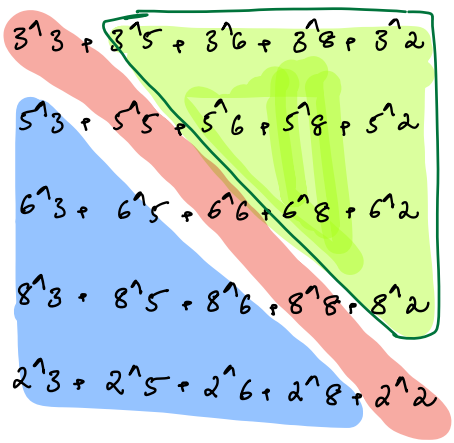
$a = \frac{V_1 + \sqrt{2V_2 - V_1^2}}{2}$

$b = a - V_1$

4Q) Given ar[N], calulate sum of xor of all pairs ? → {Google}

ar[5] = { 3 5 6 8 2 }

All pairs:



Idea: Calulate sum of xor of all pairs

```
Sum = 0
i = 0; i < n; i++) {                TC: O(N^2)
    j = 0; j < n; j++) {
        Sum = Sum + ar[i] ^ ar[j]
    }
}
return sum
```

Small optimization: TC: O(N^2) SC: O(1)

```
Sum = 0
i = 0; i < n; i++) {
    j = i+1; j < n; j++) {
        Sum = Sum + ar[i] ^ ar[j]
    }
}
return 2* sum
```

5 + 7 + 14 = 26

	2^3	2^2	2^1	2^0
5 :	0	1	0	1
+				
7 :	0	1	1	1
+				
14 :	1	1	1	0
	1	3	2	2
	2^3	$3*2^2$	$2*2^1$	$2*2^0$

$= 2^2 + 2^0 + 2^2 + 2^1 + 2^0 + 2^3 + 2^2 + 2^1$

$2^3 + 3*2^2 + 2*2^1 + 2*2^0 = 26$

$8 + 12 + 4 + 2 = 26$

3 2 1 0

2 : 0 0 1 0

3 2 1 0

3 : 0 0 1 1

3 2 1 0

5 : 0 1 0 1

3 2 1 0

6 : 0 1 1 0

3 2 1 0

8 : 1 0 0 0

	3	2	1	0
3^5 :	0	1	1	0
3^6 :	0	1	0	1
3^8 :	1	0	1	1
3^2 :	0	0	0	1
5^6 :	0	0	1	1
5^8 :	1	1	0	1
5^2 :	0	1	1	1
6^8 :	1	1	1	0
6^2 :	0	1	0	0
8^2 :	1	0	1	0

Sum = $4 * 2^3$ + $6 * 2^2$ + $6 * 2^1$ + $6 * 2^0$

Sum = 32 + 24 + 12 + 6

Sum = 74

return 2*sum = 148

Idea: In Every bit pos:

[Calculate no: of set bits in all nr pairs

TC: 32 * [O(N^2)] = $32N^2$

a^b = 1 : ⇒ { If a & b are differently

In how many arr[] elem 0th bit set =

Set: 3, 5 — unset: 2, 6, 8

No: of nr pairs with 0th bit set = 6 pairs

In how many arr[] elem 1th bit set =

Set: 3 — unset: 2

2, 3, 6 — 5, 8

No: of nr pairs with 1th bit set : 6

In how many arr[] elem 2th bit set =

Set: 2 — unset: 3

No: of nr pairs with 2th bit set : 6

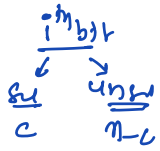
In how many arr[] elem 3th bit set =

Set: 1 — unset: 4

8 — 2, 3, 5, 6

No: of nr pairs with 3th bit set : 4

Idea:



Repeat for all bit positions:

No: of ar[] elements with i^{th} bit set = c

# of nr pairs with i^{th} bit set = $c*(n-c)$

Total Contribution i^{th} bit sum = sum + $c*(n-c)*2^i$

2^i → Base value of i^{th} bit

TC: $32*N \Rightarrow O(N)$

```
int sumxor(int ar[n]){ TC: 32*N → O(N)  SC: O(1)
    int sum = 0
    i=0; i<32; i++){
        // Calculate, no: of ar[], for which ith bit set
        c=0
        j=0; j<n; j++){
            if(checkbit(ar[j], i) == True){ c=c+1}
        }
        // No: of nr pairs with ith bit = c*(n-c)
        // Contribution of ith bit = c*(N-c)(2^i)
        sum = sum + c*(N-c)(1<<i)
                                  2^i
    }
    return 2*sum
}
```

Subarray : continous part of an array : $\frac{N*(N+1)}{2}$ subarrays

Subsequence: A sequence obtained deleting none a more elements from array, is subsequence

→ Data should be arranged based on increasing order index

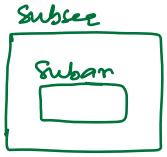
→ Empty subsequence is valid : { }

	0	1	2	3	4	5		subseq:
ar[6] =	3	2	1	9	6	8		
	✓	*	✓	x	*	✓	→	3 1 8
	*	x	✓	✓	✓	✓	→	8 6 1 9 * order is must
								1 9 6 8
	*	✓	*	✓	✓	✓	→	2 9 6 8
	*	*	*	*	*	*	→	{ }
	*	✓	✓	*	✓	*	→	{2 1 6}
	✓	✓	✓	✓	✓	✓	→	{3 2 1 9 6 8}

obs:

→ Every subsequence is subarray : No

→ Every subarray is subsequence : Yes



ar[3] = { 3 1 8 } —sort arr[]→ ar[] = {1 3 8}

Sum	Subsequence:	Subsequence:	sum	max	min
0	{ }	{ }	0		
3	{3}	{3}	3		
1	{1}	{1}	1		
8	{8}	{8}	8		
4	{3 1}	{1 3}	4		
9	{1 8}	{1 8}	9		
11	{3 8}	{3 8}	11		
12	{3 1 8}	{1 3 8}	12		

order, changing, data same

order changing, data same

Obs: When we sort arr[] subsequence will change

↳ // Questions revolving around subsequences not related to sort

Q] Sum of every subsequence
Max of every subsequence
Min of every subsequence

} Sorting arr[] is still okay for us, because data is not changing in subsequence

Note: Given ar[N] no: of subsequence = 2^N

↳ Including { } sequence

Q) Given ar[N], check if there exists a subsequence with sum = k

ar[8] = { 6 8 3 2 4 1 9 10 }

k = 18 → { 8 1 9 } { 8 10 } .... {return True}

k = 30 → { 6 8 2 4 10 } ... {return True}

Ideas: Generate all subsequences & calculate their sum == k

Ex: ar[3] = { 9 14 10 } : {0, 7} =

(indices: 0 1 2)

num:	Binary (2 1 0)	subseq
0	0 0 0	→ { }
1	0 0 1	→ {9}
2	0 1 0	→ {14}
3	0 1 1	→ {9 14}
4	1 0 0	→ {10}
5	1 0 1	→ {9 10}
6	1 1 0	→ {14 10}
7	1 1 1	→ {9 14 10}

3 numbers → 8 subseq

3 bits → 8 3 bit numb

mapped every 3 bit number to a subsequence

1st 3 bit: 0 0 0 → 0

⋮

last 3 bit: 1 1 1 → 7

All ar[N] all subseq = 2^n

n bit numbers → 2^n

every n bit number → subsequence

n-1 ... 2 1 0

0 ... 0 0 0 → 0 → ⋯

1 ... 1 1 1 → $2^n - 1$ → ⋯

```
void printsub(int ar[]){ TC: 2^N * N = O(N*2^N)  SC: O(1)

    for(i=0; i<2^n; i++){        constraints N<=20
        // i is an N bit number → subsequence
        for(j=0; j<n; j++){
            // consider
            if(checkbit(i,j)) { print(ar[j]); }
        }
        // an entire subsequence done
        print("newline")
    }
}
```

ar[3] = { 9 14 10 } (0 1 2)

2 1 0	i	j: 0	j: 1	j: 2	output
0 0 0	0	*	*	*	{ }
0 0 1	1	✓	*	*	{9}
0 1 0	2	*	✓	*	{14}
0 1 1	3	✓	✓	*	{9 14}
1 0 0	4	*	*	✓	{10}
1 0 1	5	✓	*	✓	{9 10}
1 1 0	6	*	✓	✓	{14 10}
1 1 1	7	✓	✓	✓	{9 14 10}

```
bool subsub(int ar[N], int k){ TC: O(N*2^N) SC: O(1)
    for(i=0; i<2^n; i++){
        // i is an N bit number → subsequence
        sum = 0
        for(j=0; j<n; j++){          // check if value i has j^th bit set
            if(checkbit(i, j)){ sum = sum + ar[j]; }   // consider
        }
        if(sum == k){ return True; }
    }
    return false
}
```

Note: ar[N] subseq with sum = k

- 1) Bit manipulations ⇒ $(N*2^N)$
- 2) Back Tracking ⇒ (2^N) * {later}
- 3) Dynamic Programming ⇒ $O(N*k)$ {later}

Subset: It is similar to subsequence, order doesn't matter

	0	1	2	3	4	5	subseq:
ar[6] =	3	2	1	9	6	8	
	✓	*	✓	x	*	✓	→ 3 1 8
	*	x	✓	✓	✓	✓	→ 8 6 1 9 ✓ } Both are 1 9 6 8 ✓ } same

ar[3] = { 3 1 8 } —Sort arr[]→ ar[] = {1 3 8}

Subset

{ }
{3}
{1}
{8}
{3 1}
{1 8}
{3 8}
{3 1 8}

Subset

{ }
{3}
{1}
{8}
{1 3}
{1 8}
{3 8}
{1 3 8}

Obs: In case of subsets we can sort arr[], it won't effect subsets, because order not matter

↳ Generating subsets, we can do it above bit manipulation technique

↳ **TC:** O(N*2^N)