Single Number II (repeat no. repeat 3 times)

[5, 5, 5, 6, 4, 4, 4]

	2	0	1
5 →	1	0	1
5 →	1	0	1
5 →	1	0	1
6 →	1	1	0
4 →	1	0	0
4 →	1	0	0
4 →	1	0	0
Total bit cnt	7	1	3
per index	%3	%3	%3
	1	1	0

Observation

Each repeater contributor contributes set 1 bits / index in multiples of 3.

So if no. single number existed, all total bit cnt/index would be multiples of 3.

This logic helps out find the extra bits & recreate the number

Method 1

works for +ve only

assume int range

```
ans = 0
for (bit2nd := 0 → 31) {
    cnt = 0
    for (i := 0 → (n-1)) {
        if (nums[i] & (1 << bit2nd)) cnt++;   // bit2nd bit of nums[i] is set
    }
    if (cnt % 3 == 1) {
        ans = ans | (1 << bit2nd);            // turn on bit2nd bit of ans
        if (bit2nd < 31) else ans -= (1<<31)  // handles -ve
    }
}
return ans;
```

T.C = O(32n)

S.C = O(1)

Method 2

sort + observation [2 2 1 2 1 1 4 3 4 4] → sort → [1 1 1 2 2 2 3 4 4 4]

groups of 3

```
Sort (nums);
for (i = 1; i < n; i = i+3) {
    if (nums[i] != nums[i-1]) {
        return nums[i-1];
    }
}
return nums[n-1];
```

T.C = O(N log n + n/3)

S.C = O(1)

State Machine method

Goal → (Count the number of '1' for bit position in all no.s) % 3

Hence our ~~domain~~ range of result lies within 0, 1, 2 — finite no. of states.

Appearance count	↳ Bit State (mod 3)	Stored in ones, twos (c_1, c_2)
0 (not seen)	0	0, 0
1 (seen once)	1	1, 0
2 (seen twice)	2	0, 1
3 (seen thrice)	0 (reset)	0, 0

} kind of mod-3 counter

This is basically the same approach as method 1 but instead of 32 checks for number we compute all bits in one pass.

Its like State 0 → State 1 → State 2 → State 0 — ...

Logically, we are trying to eradicate all numbers that exist thrice as not seen & only 1 number is saved to state 1.

The thing we are trying to achieve is :-

Nums list	Exist once	Exist twice	Exist thrice (vanish)	Storage in single var. ones	twos
x	✓			just x	
z	✓			Combo (x, z)	
y	✓			Combo (x, y, z)	
z		✓		Combo (x, y)	just z
x		✓		just y	Combo (z, x)
x			✓		just z
z			✓		<empty>

y exist once

This only works when just 1 number is single & others have freq. 3.

Writing combo ain't fair either as bit level operations show results at end

We cannot extract numbers in middle of operation.

We will try to derive boolean logic for ones (1 occurence) (let c_1) & twos (2 occurences) (let c_2).

Note :- We are talking in terms of ith bit of num[x] & boolean operation in ones, twos compute all 32 bits

num[x] → { ... 101 ... [i] ... 010 ..... }
ith bit

old two (c_2)	old one (c_1)	i^{th} bit	new two (c_2')	new one (c_1')	
0	0	0	0	0	no change
0	0	1	0	1	
0	1	0	0	1	no change
0	1	1	1	0	
1	0	0	1	0	no change
1	0	1	0	0	RESET (3 occurences)
1	1	0	X	X	Not Reachable
1	1	1	X	X	" "

Applying SOP logic,

$$c_1' = \bar{c_2}\,\bar{c_1}\,i + \bar{c_2}\,c_1\,\bar{i}$$

$$\Rightarrow c_1' = \bar{c_2}\,(\bar{c_1}\,i + c_1\,\bar{i})$$

$$\Rightarrow \boxed{c_1' = \bar{c_2}\,(c_1 \oplus i)} \quad \cdots (i)$$

$$\boxed{c_2' = \bar{c_2}\,c_1\,i + c_2\,\bar{c_1}\,\bar{i}} \quad \cdots (ii)$$

But, if we consider new c_1 ie. c_1' instead.

$$c_2' = \bar{c_2}\,\bar{c_1'}\,i + c_2\,\bar{c_1'}\,\bar{i}$$

$$\Rightarrow c_2' = \bar{c_1'}\,(\bar{c_2}\,i + c_2\,\bar{i})$$

$$\Rightarrow \boxed{c_2' = \bar{c_1'}\,(c_2 \oplus i)} \quad \cdots (iii)$$

Let try to prove both formulas indeed give same result :-

$$c_2' = \overline{\{\bar{c_2}\,(\bar{c_1}\,i + c_1\,\bar{i})\}}\,(\bar{c_2}\,i + c_2\,\bar{i}) \quad [\text{Put } (i) \text{ in } (iii)]$$

$$= \{c_2 + \overline{(\bar{c_1}\,i + c_1\,\bar{i})}\}\,(\bar{c_2}\,i + c_2\,\bar{i}) \quad [\text{De Morgan}]$$

$$= \{c_2 + (\overline{\bar{c_1}\,i} \cdot \overline{c_1\,\bar{i}})\}\,(\bar{c_2}\,i + c_2\,\bar{i}) \quad [\text{D...}]$$

$$= \{c_2 + (c_1 + \bar{i})(\bar{c_1} + i)\}\,(\bar{c_2}\,i + c_2\,\bar{i})$$

$$= \{c_2 + c_1\bar{c_1} + c_1 i + \bar{i}\bar{c_1} + \bar{i}\,i\}\,(\bar{c_2}\,i + c_2\,\bar{i})$$

$$= c_2\bar{c_2}\,i + c_2 c_2\,\bar{i} + c_1 i\,\bar{c_2}\,i + c_1 i\,\bar{c_2}\,\bar{i} + \bar{i}\bar{c_1}\bar{c_2}\,i + \bar{i}\bar{c_1}c_2\,\bar{i}$$

$$= c_2\,\bar{i} + \bar{c_2}c_1 i + c_2\bar{c_1}\,\bar{i}$$

$$= c_2\,\bar{i}\,(1 + \bar{c_1}) + \bar{c_2}c_1 i = c_2\,\bar{i} + \bar{c_2}c_1 i$$

$\therefore\ c_2' = c_2\bar{i} + \bar{c_2}c_1 i$

But from (ii), $c_2' = \bar{c_2}c_1 i + c_2\bar{c_1}\bar{i}$ (missing)

Let check values via truth table,

c_2	c_1	ith bit	$c_2\bar{c_1}\bar{i}$	$c_2\bar{i}$
0	0	0	$(0\cdot1\cdot1)=0$	$(0\ 1)=0$
0	0	1	$(0\cdot1\cdot0)=0$	$(0\ 0)=0$
0	1	0	$(0\cdot0\cdot1)=0$	$(0\ 1)=0$
0	1	1	$(0\ 0\ 0)=0$	$(0\ 0)=0$
1	0	0	$(1\ 1\ 1)=1$	$(1\ 1)=1$
1	0	1	$(1\ 1\ 0)=0$	$(1\ 0)=0$
unreachable states. 1	1	0	$(1\ 0\ 1)=0$	$(1\ 1)=1$
1	1	1	$(1\ 0\ 0)=0$	$(1\ 0)=0$

For these $(c_2\bar{c_1}\bar{i}) = (c_2\bar{i})$

Don't care

Mathematically $(c_2\bar{c_1}i)$ & $(c_2\bar{i})$ are different but when paired with the Don't care states they are logically equivalent.

Except for Don't care states, we see for all cases $(c_2 = \bar{c_1})$

So we merge both to get $(c_2\bar{i})$

Assume num = [2, 2, 3, 2]

num [x]	c_1 new $= \overline{c_2}$ old $(c_1$ old $\oplus$ num)	c_2 new $= \overline{c_1}$ new $(c_2$ old $\oplus$ num)
	0 (Initially)	0 (Initially)
2 $(10)_2$	$\overline{00}\ (00 \wedge 10) = 11\cdot10 = (10)_2 \rightarrow 2$	$\overline{10}\ (00 \wedge 10) = 01\cdot10 = (00)_2 \rightarrow 0$
2 $(10)_2$	$\overline{00}\ (10 \oplus 10) = 11\cdot00 = (00)_2 = 0$	$\overline{00}\ (00 \oplus 10) = 11\cdot10 = (10)_2 = 2$
3 $(11)_2$	$\overline{10}\ (00 \oplus 11) = 01\cdot11 = (01)_2 = 1$ (Neither 2 nor 3, a substate value) bit level working	$\overline{01}\ (10 \oplus 11) = 10\cdot01 = 0$
2 $(10)_2$	$\overline{00}\ (01 \oplus 10) = 11\cdot11 = (11)_2 = 3$ the number with single occurence	$\overline{11}\ (00 \oplus 10) = 00\cdot10 = (00)_2 = 0$

Code 1 ← involves old c_1 (ones) & old c_2 (twos)

```
int SingleNumber (vec<int> &nums) {
        int c1_new = 0, c2_new = 0;
        for (int num : nums) {
            int c1_old = c1_new, c2_old = c2_new;
            c1_new = ~c1_old & (num ^ c1_old);
            c2_new = (num & c1_old & (~c2_old))
                   | ((~num) & (~c1_old) & c2_old);
        }
        return c1_new;
}
```

Code 2 ← involves use of c_1' i.e. new c_1 to compute c_2

```
int singleNumber (vec<int>&nums ){
    int c1=0, c2=0;
    for (int num : nums) {
        c1 = (c1 ^ num) & ~c2;
        c2 = (c2 ^ num) & ~c1;
    }
    return c1;
}
```

this c1 is newer updated c1

T.C = O(n) where n = no. of elements in array

S.C = O(1)