BitMap

15 June 2022 20:36

AlgoU Session 4 B...

Alex and Sam are very good friends. Alex is doing programming a lot these days. He has set a target A for himself. Initially, Alex score is zero. Alex can only double his score by doing a question or Alex can seek help from Sam for doing questions which will contribute 1 to Alex score. Alex wants his score to be exactly A. Also, he does not want to take much help from Sam.
Find and return the minimum number of times Alex needs to take help from Sam to achieve a score of exactly A.

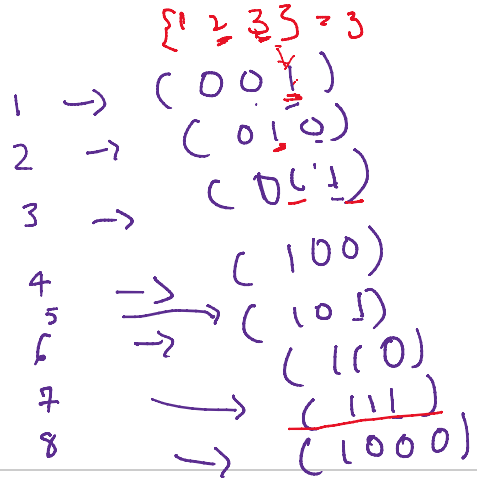
0 <= A <= 10^9

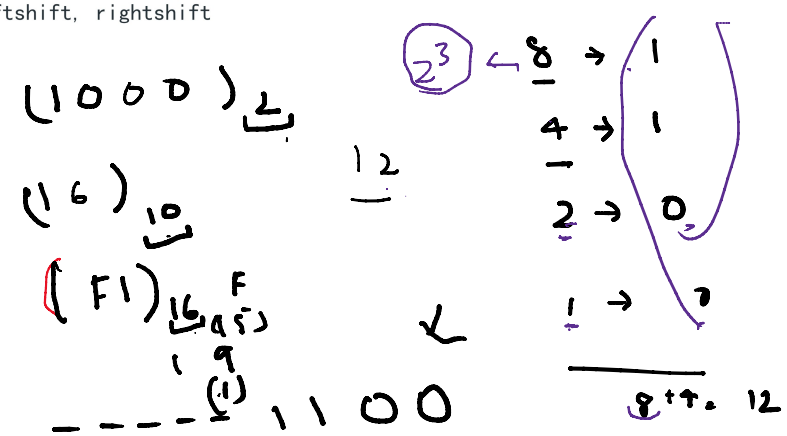
Input 1:
A = 5

Output 1:
2

Bit Manipulation Background

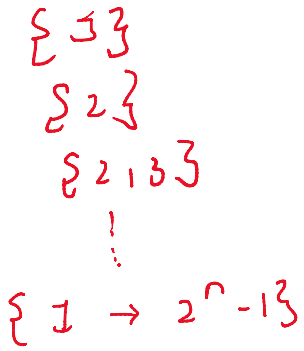
- Representing numbers as binary
- When is representing nos. as binary useful?
- Iterate through all subsets of {1, 2, 3, 4}
- Bitwise operations - OR , AND, XOR, leftshift, rightshift

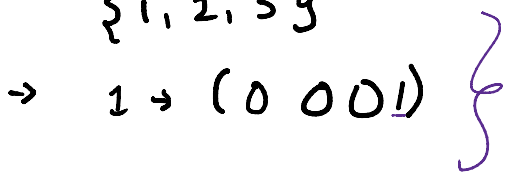




Bit Manipulation Background

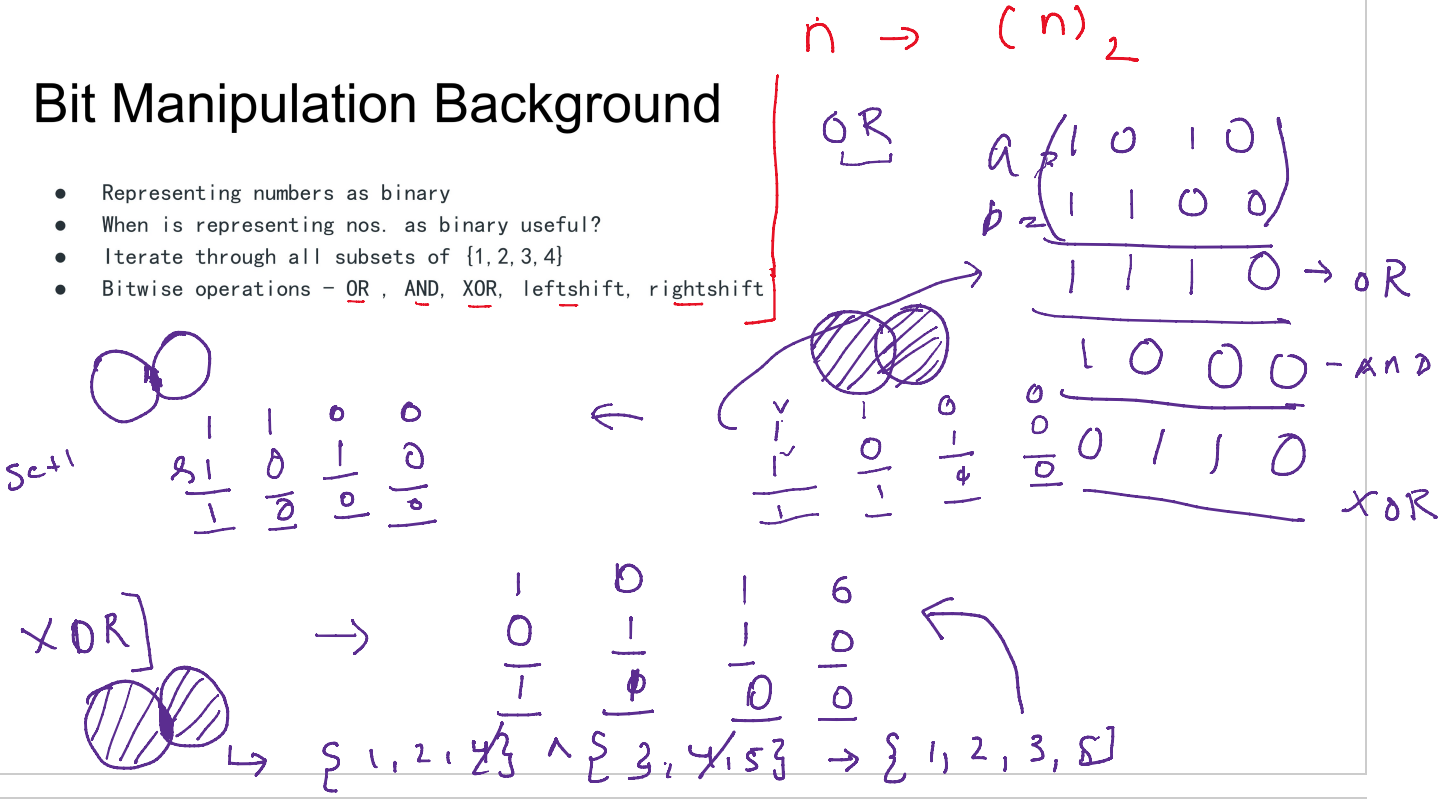
- Representing numbers as binary
- When is representing nos. as binary useful?
- Iterate through all subsets of {1, 2, 3, 4}
- Bitwise operations - OR , AND, XOR, leftshift, rightshift





Bit Manipulation Background

- Representing numbers as binary
- When is representing nos. as binary useful?
- Iterate through all subsets of {1, 2, 3, 4}
- Bitwise operations - OR , AND, XOR, leftshift, rightshift



Introduction

Bitwise Operations are faster and closer to the system and sometimes optimize the program to a good level.

1 byte comprises of 8 bits and any integer or character can be represented using bits in computers, which we call its binary form(contains only 1 or 0) or in its base 2 form.

Introduction

Bitwise Operations are faster and closer to the system and sometimes optimize the program to a good level.

1 byte comprises of 8 bits and any integer or character can be represented using bits in computers, which we call its binary form(contains only 1 or 0) or in its base 2 form.

Example:

1) 14 = $\{1110\}_2$

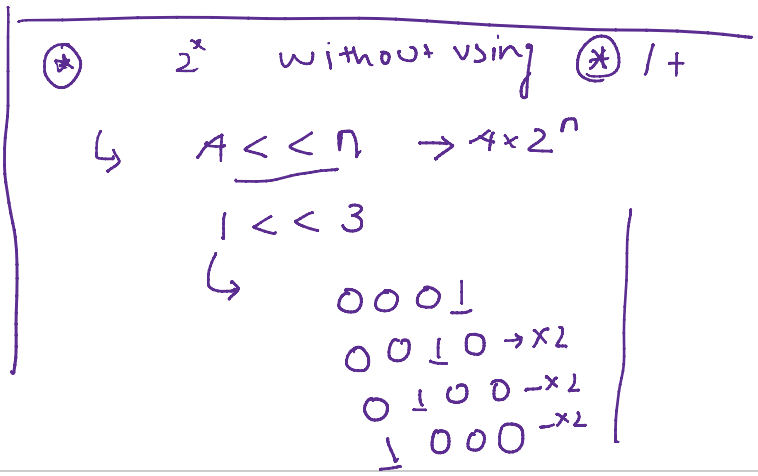
= $1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$

= 14.

2) 20 = $\{10100\}_2$

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

= 20.



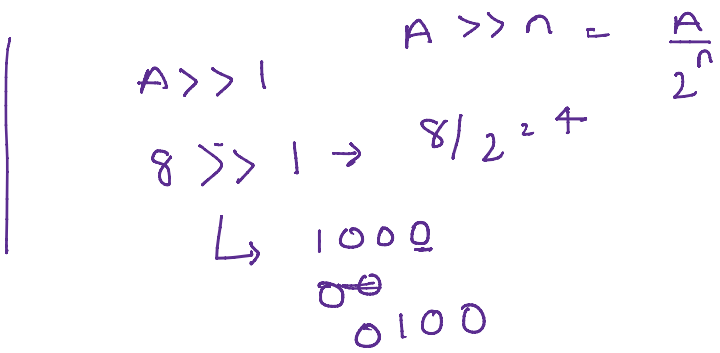
Bitwise Operators

NOT (~):

AND (&)

OR (|)

XOR (^)



Left Shift (<<): Left shift operator is a binary operator which shift the some number of bits, in the given bit pattern, to the left and append 0 at the end. Left shift is equivalent to multiplying the bit pattern with 2^k (if we are shifting k bits).
$1 << n = 2^n$
Right Shift (>>): Right shift operator is a binary operator which shift the some number of bits, in the given bit pattern, to the right and append 1 at the end. Right shift is equivalent to dividing the bit pattern with 2^k (if we are shifting k bits).
16 >> 4 = 1

Accessing ith bit

x &(1<<i)

→ for (i → length)

if ind i^{th} bit

i^{th} bit of given number

binary (given number)

0000 ... 1 000

↓

000 1 00 ... 0 (and)

0 0 !

↓

O(n)

How to check if a given number is a power of 2 ?

(obv)

O(1) 5 mins

n & (n-1)

→ convert binary

→ set bit

→ O(32)

O(64)

→ O(1/2)

→ n = 0 ?

→ true 2's

N.A

binary

↓

set bits

↓

Bit map

↓

Unsig

↓

O(1)

Sing ir

+

$\log_a^x$

$\log_a a^x \rightarrow x$

→ !(n & (n-1))

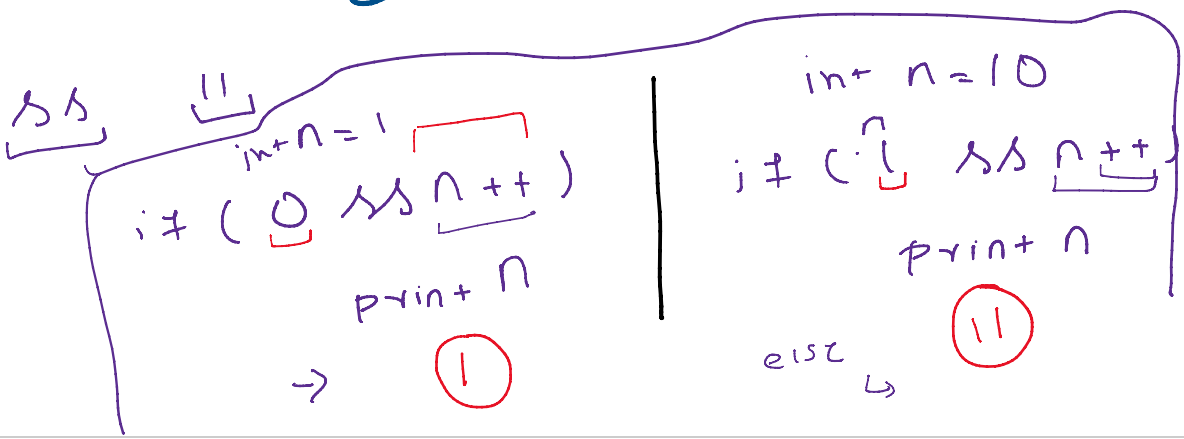
→ n → 16 → 10000

15 → 01111

00000

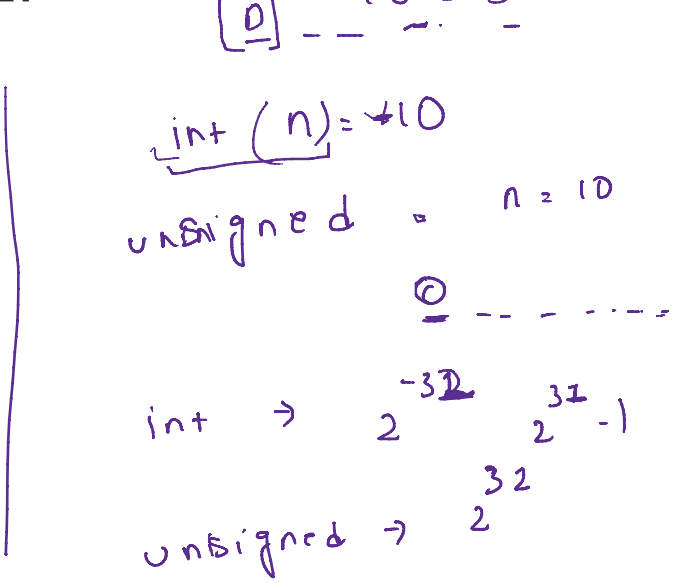
2^x → 1 set bit

!. n & (n-1))



How to check if a given number is a power of 2 ?

```
bool isPowerOfTwo(int x){
    if(x == 0)  return false;
    else {
        while(x % 2 == 0) x /= 2;
        return (x == 1);
        }
  }
```



How to check if a given number is a power of 2 ?

```
bool isPowerOfTwo(int x)
  {
    // x will check if x == 0 and !(x & (x - 1)) will check if x is a power of 2 or not
    return (x && !(x & (x - 1)));
  }
```

(__builtin_popcount(x) / __builtin_popcountll(x)) 2 → 1, 3 → 2

↳ count set bits

Quickest way to swap two numbers

* XOR

```
a ^= b;
b ^= a;
a ^= b;
```

$P_1 \rightarrow a \wedge a = 0$

$P_2 \rightarrow a \wedge 0 = a$

$a' = a \wedge b$

$b = b \wedge a'$

$= b \wedge a \wedge b$

$= b \wedge b \wedge a$

$= 0 \wedge a$

$b = a$

$a = c = a$

$a = b$

$b = c$

$a = 10 \quad b = 1$

$a = 11$

$b = 11 - 1 = 10$

$a = 11 - 10$

$= 1$

$a = 1 \quad b = 10$

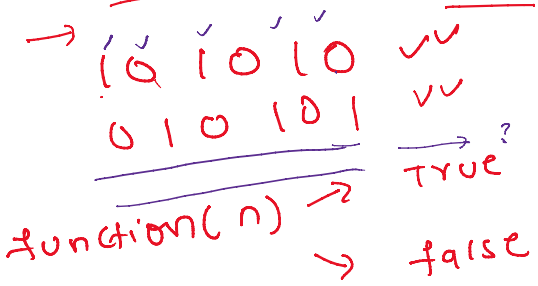
$a = a + b$

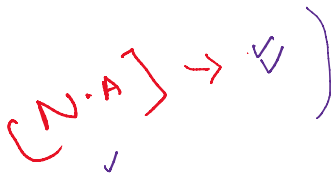
$b = a - b$

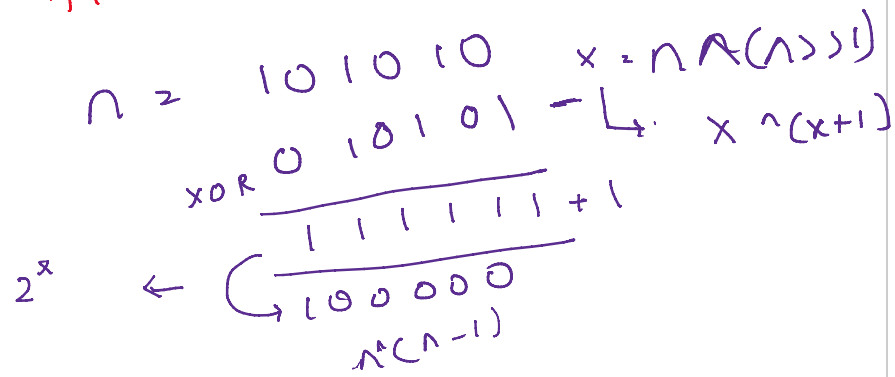
$a = a - b$

Check if a number has bits in alternate pattern

5 min





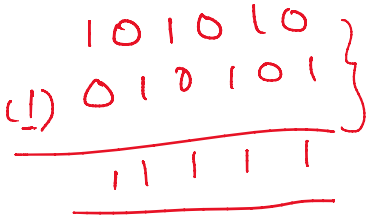


Check if a number has bits in alternate pattern

```
bool allBitsAreSet(unsigned int n)

{

    if (((n + 1) & n) == 0) return true;

     return false;

}

bool bitsAreInAltOrder(unsigned int n)

{

    unsigned int num = n ^ (n >> 1);

return allBitsAreSet(num);

}
```

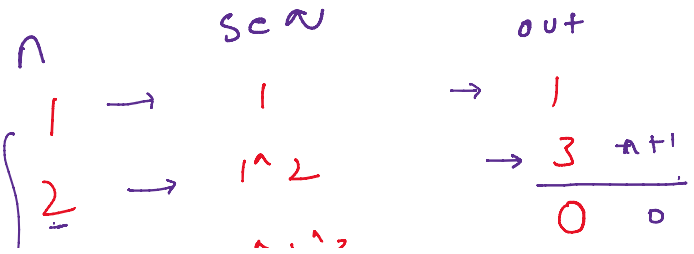
→ check if all set

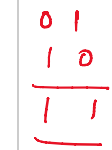
can we replace XOR with &|

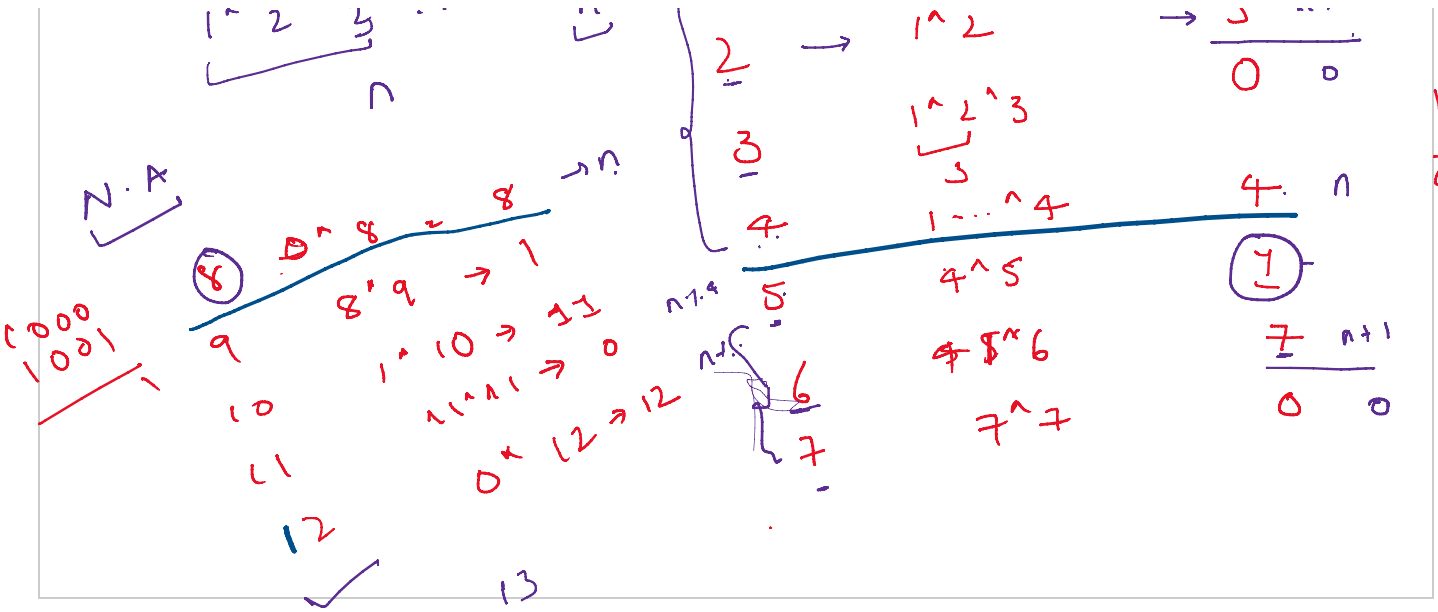


→ 1111 - all set

Compute XOR from 1 to n







Compute XOR from 1 to n

```
int computeXOR(int n)
{
    if (n % 4 == 0) return n;
    if (n % 4 == 1) return 1;
    if (n % 4 == 2) return n + 1;
    else
        return 0;
}
```

Compute XOR from 1 to n

```
int computeXOR(int n)
{
    if (n % 4 == 0) return n;
    if (n % 4 == 1) return 1;
    if (n % 4 == 2) return n + 1;
```

```
if (n % 4 == 1) return 1;
if (n % 4 == 2) return n + 1;
    .
    .
    .
else
    return 0;
}
```

case(i) $n \% 4 = 0$ = $1 \wedge 2 \wedge 3 \wedge \dots \wedge n-1 \wedge n$

$T(n-1 \% 4 = 3) \wedge n$

$0 \wedge n$

$= n$

$T(n \% 4 = 2 \text{ or } 1) \mid 1 \wedge 2 \dots \wedge n-1 \wedge n$

$T(n-1 \% 4 = 0) \wedge n$

n odd

$n-1 \rightarrow$ even

$6 \rightarrow 110$

$7 \rightarrow 111$

$(n-1) \wedge n$

$= 1$

$1 \wedge 0 = 1$

$1 \wedge 1 = 0$

$0 \wedge 1 = 1$

$(n \% 4 == 2) = T((n-1) \% 4 = 1) \wedge n$

$1 \wedge n$

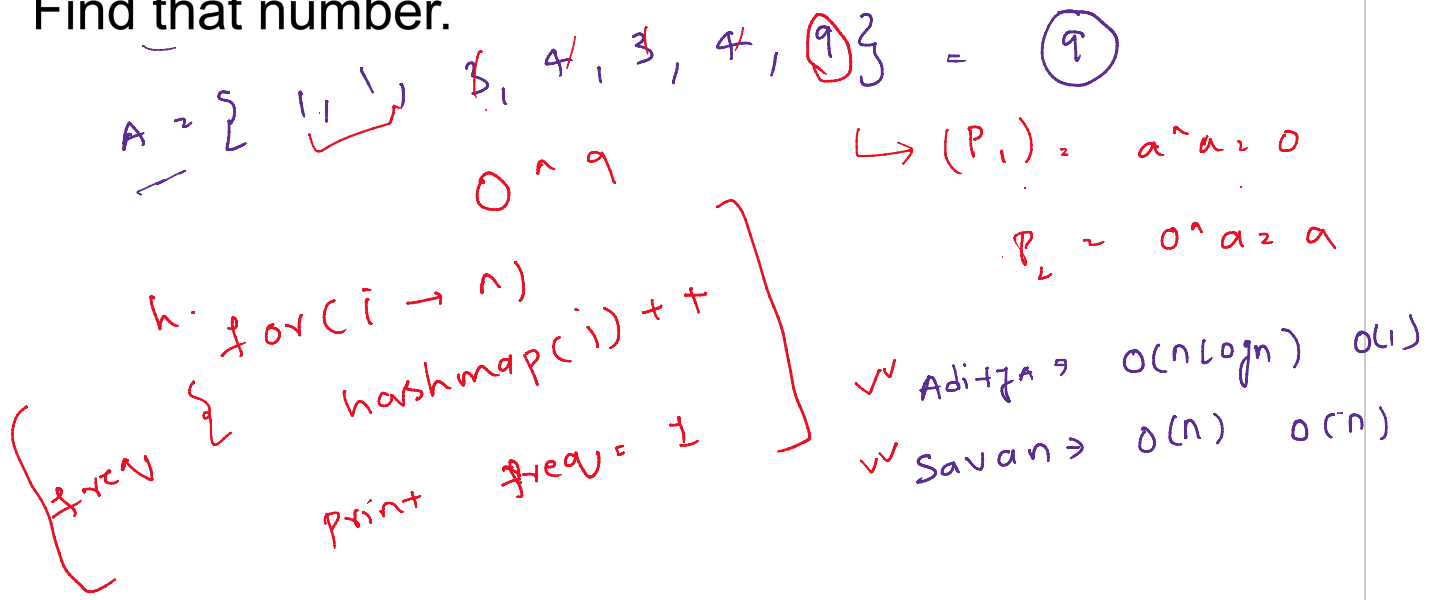
$= n+1$

$n \% 4 = 3 = T(n-1 \% 4 = 2) \wedge n$

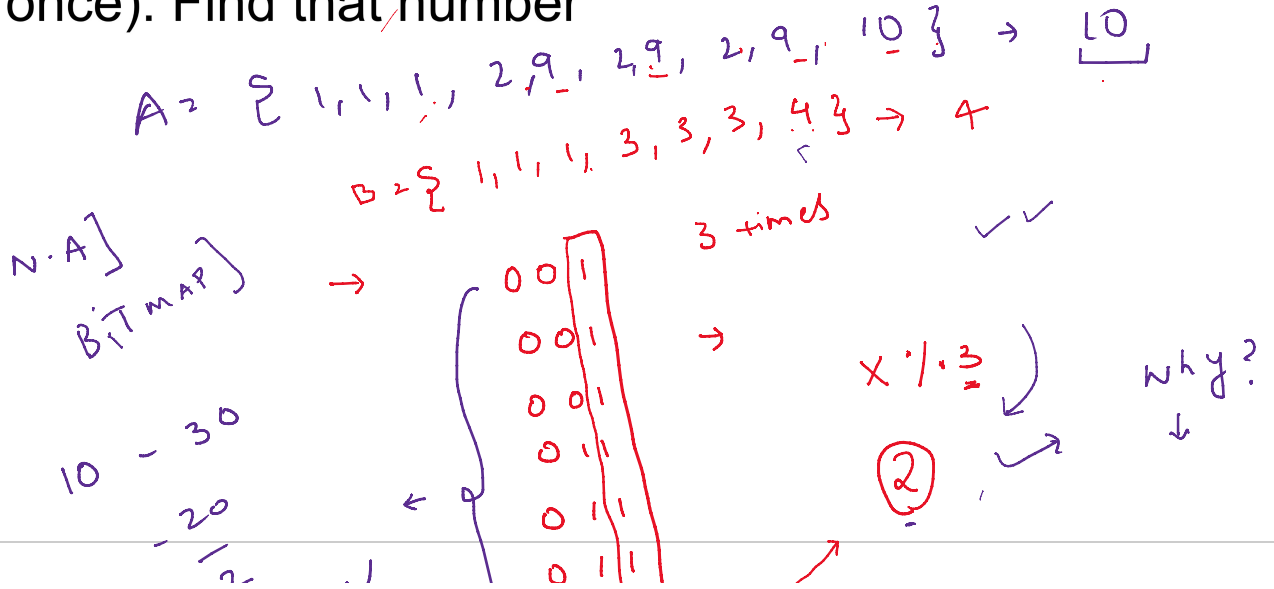
$= n \wedge n$

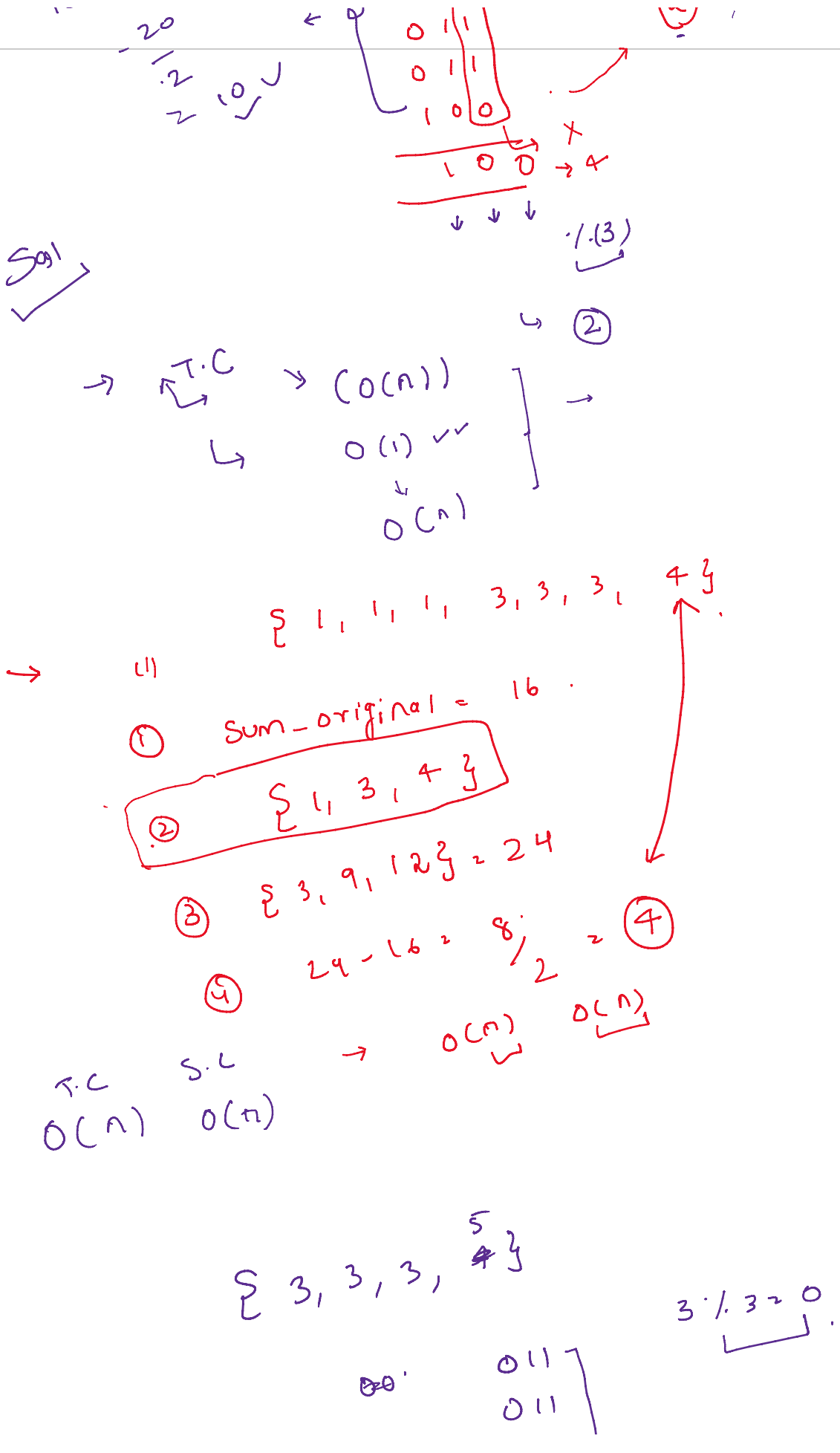
$= 0$

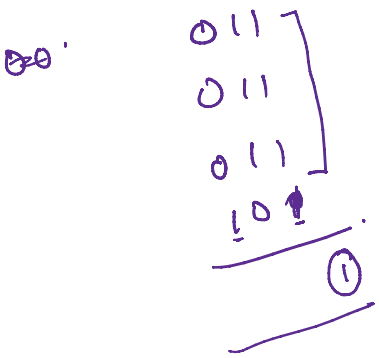
* In an array, every number appears twice except one. Find that number.



* You are given an array in which every number appears thrice except one(which occurs exactly once). Find that number




20
2
10
0 1 1
0 1 1
1 0 0
1 0 0 → 4
Sol
T.C
O(n)
O(1)
O(n)
{ 1, 1, 1, 3, 3, 3, 4 }
(1)
① Sum - original = 16
② { 1, 3, 4 }
③ { 3, 9, 12 } = 24
④ 24 - 16 = 8 / 2 = 4
O(n)
O(n)
T.C
S.C
O(n)
O(n)
{ 3, 3, 3, 5 }
3 % 3 = 0
0 1 1
0 1 1



→ Each Element rep^{n} 5 times, 1 element

→