→ Bit Manipulation

↳ refers to the act of algorithmically modifying bits (0s & 1s) within an Integer using bitwise operator.

→ Binary number conversion

num = 32

2	32	
2	16	0
2	8	0
2	4	0
2	2	0
	1	0

→

```
StringBuilder sb = new SB();
while(num > 0){
    sb.append(num%2);
    num = num/2
}
return sb.reverse().toString();
```

→ Binary to decimal

1 0 0 0 0 0

$1\times2^5 + 0\times2^4 + 0\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0$

→ 1's & 2's → complement

1's → Flip all the bit

1 0 0 0 0 0

0 1 1 1 1 1 → Flip

2's → Do 1's complement & add 1 to it

0 1 1 1 1 1

\+ 1

1 0 0 0 0 0

Part1.java

```java
public static @NotNull String toBinary(int num) {
    if (num == 0) return "0";
    StringBuilder sb = new StringBuilder();
    while (num > 0) {
        sb.append(num % 2);
        num = num / 2;
    }

    return sb.reverse().toString();
}
public static int toDecimal( @NotNull String binary){
    int res = 0;
    int power = 1;

    for (int i = binary.length() - 1; i >= 0; i--) {
        char ch = binary.charAt(i);

        if (ch != '0'){
            res += power;
        }
        power = power * 2;
    }
    return res;
}
```

TC: $O(\log_2 N)$

SC: $O(\log_2 N)$

→ Binary - operators

- **AND (&)**

a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

- **OR (|)**

a	b	a|b
0	0	0
0	1	1
1	0	1
1	1	1

- **XOR (^)**

a	b	a^b
0	0	0
0	1	1
1	0	1
1	1	0

- even no. of 1 → 0
- odd no of 1 → 1

- **Right shift (>>)**

n = 13 >> 1
↳ right shift the bit by 1

1 1 0 1 → right shift
→ 1 1 0 = 6

$x >> k = \frac{x}{2^k}$

- **Left-shift (<<)**

13 << 1

1 1 0 1
1 1 0 1 0 → 26

$x << k = x * 2^k$

↓
what if we do ?
↓
$(2^{31} - 1) << 1$

0 1 1 . . . . 1 1 1
1 1 . . . . 1 1 0
↳ one of Integer overflows

- **NOT (~)**

$x = \sim(-6)$

1. Flip
2. Check -ve
yes → 2's complement
stop

$x = \sim(5)$

⇒ 0 0 0 . . . . 1 0 1
1 1 1 . . . . 0 1 0 → Flip
-ve ← 1
↳ Takes 2's complement
1 0 0 . . . . 1 0 1
+1
1 0 0 . . . . 1 1 0 → -6

→ Swap 2 numbers without third variable

$a = a \wedge b$

$b = a \wedge b \rightarrow \{ a \wedge b \wedge b = a \}$

$a = a \wedge b \rightarrow \{ a \wedge b \wedge a = b \}$

↳ $b = a$ & $a = b$

★ $a \wedge 0 = a$

$a \wedge a = 0$

→ Check if ith bit is set or not

num = 34 i = 3

100010 >> i

100

& 1

0

if it is 0, then ith bit is not set else i.e if it is 1 then ith bit is set

(num >> i) & 1 == 1

(either return true or false.

→ Set the ith bit ⇝ make ith bit 1

num = 34 i = 3

a 100010

00001 << i

b 000100

a|b 100110

ith bit set

→ Clear the ith bit make ith bit 0

num = 34, i = 5

a 100010

00001 << i

b 100000

~b 011111

a & ~b 000010

num & (~(1<<i))

num | (1 << i)

→ Toggle the ith bit → make 1→0 & 0→1

num = 34 i = 2

```
a   1 0 0 0 1 0
    0 0 0 0 1     << i
  ---------------
b   0 0 0 1 0 0
  ---------------
a^b 1 0 0 1 1 0
```

num ^ (1 << i)

→ Remove the last set-bit (Right most)

N = 16

```
N-1   10000
      01111
    & -----
      00000
```

N = 32

```
N-1   100000
      011111
    & ------
      000000
```

num & (num-1)

→ check if the number is power of two

↳ From above two examples all power of two number become 0, if we do n & n-1;

n & n-1 == 0 → yes it is power of 2

→ check if a number is even/odd

num = 16

```
  10000
& 00001
  -----
  00000
```

num = 17

```
  10001
& 00001
  -----
  00001
```

- if the number is even, the right most bit is 0, if it is odd, the right most bit is 1

n & 1 == 0

→ Count the number of set bits

n = 24

2	24	
2	12	0
2	6	0
2	3	0
2	1	1

11000 → no of set bits = 2

→ can also be written as n & 1 == 1

num % 2 == 1
↳ count++;
num = num/2;
↳ num = num >> 1

PartII.java

```java
public static int countSetBit(int num){
    int count = 0;

    while (num > 0){
        count += num & 1;
        num = num >> 1;
    }
    return count;
}
```

→ 0/1

→ Minimum bit flips to convert number

start = 10 goal = 7

10 → 1010
7 → 0111
^ ——————
1101

↳ 3 bit flips to make start → goal

we can count the set bits to find the result.

PartIII.java

```java
public static int minBitFlips(int start , int end){
    return countSetBit( num: start ^ end);
}
@Contract(pure = true)
public static int countSetBit(int num){  1 usage  new
    int count = 0;

    while (num > 0){
        count += num & 1;
        num = num >> 1;
    }
    return count;
}
```

→ power - set ⇝ generate all possible set

nums = [1,2,3] = { [], [1], [2], [3], [1,2], [2,3], [1,3], [1,2,3] }

→ 2^n

iterate → 0 ⇒ not-take
1 ⇒ Take

2	1	0	
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

2^n subset

PartIV.java

```java
public static @NotNull List<List<Integer>> pow(int @NotNull [] num) {
    List<List<Integer>> power = new ArrayList<>();

    for (int i = 0; i < 1 << num.length; i++) {
        List<Integer> list = new ArrayList<>();

        for (int j = 0; j < num.length; j++) {
            boolean isSet = ((i >> j) & 1) == 1;
            if (isSet) list.add(num[j]);
        }

        power.add(list);
    }
    return power;
}
```

→ Single Number I

↳ Given a non-empty array of Integers nums, every element appears twice except for one. Find that single one.

1	8	6	8	1	6	10

→ Answer = 10

XOR-properties

$x \wedge 0 = x$

$x \wedge x = 0$

So, $1 \wedge 8 \wedge 6 \wedge 8 \wedge 1 \wedge 6 \wedge 10$

$\Rightarrow 1 \wedge 1 \wedge 8 \wedge 8 \wedge 6 \wedge 6 \wedge 10$

$\Rightarrow 0 \wedge 0 \wedge 0 \wedge 10$

$\Rightarrow 10$

```java
// PartIV.java
public static int singleNumber(int @NotNull [] nums) {
    int xor = 0;
    for (int e : nums) xor = xor ^ e;
    return xor;
}
```

→ Single Number II

↳ every element appears twice except one. Return that no.

1	1	2	1	3	3	3

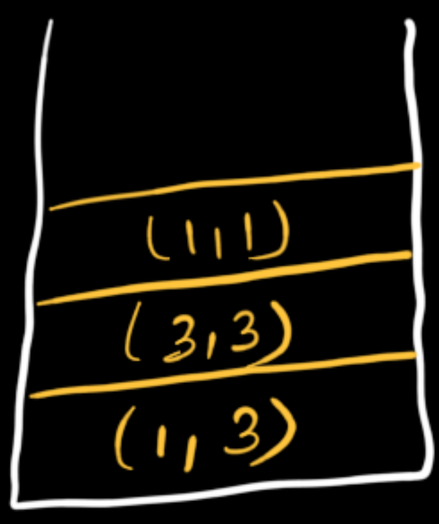
→ Approach-1

↳ • use Map to count frequencies O(N)

• Iterate over map & if at any O(N)

key → value = 1

↳ return key

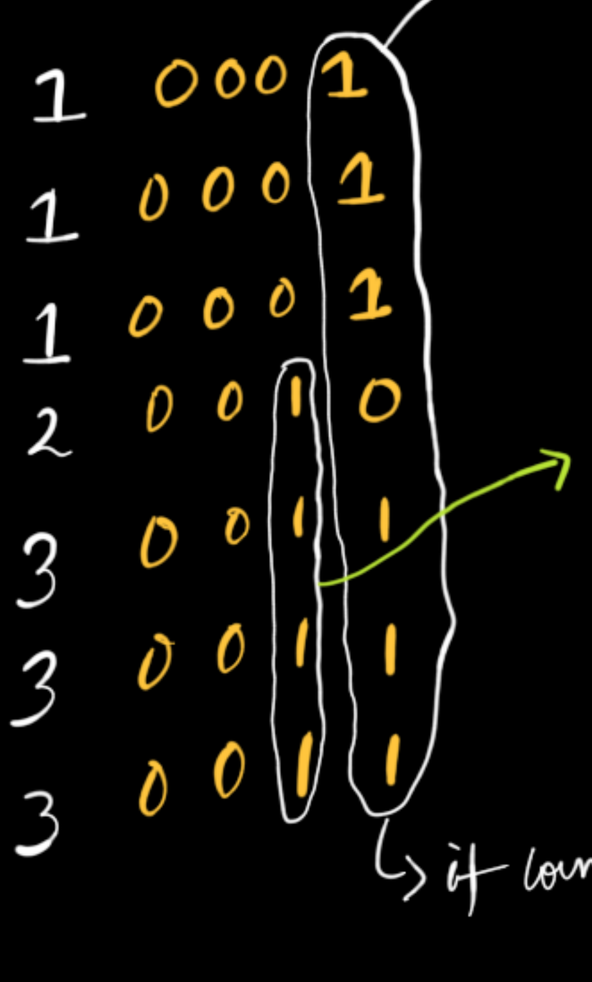


Map<nums[i], freq>

TC: ≈ O(2N) SC: O(N)

→ Approach-2

Count how many times each bit is set across all numbers



ans = 0 0 0 0

↓

1

↳ set the bit

4 set bits

↳ count % 3 == 1

↳ set that bit

↳ if count is multiple of 3

↳ must have come from repeating numbers

```java
// PartV.java
public static int singleNumberIIBetter(int[] nums) {
    int ans = 0;

    for (int bitIndex = 0; bitIndex < 32; bitIndex++) {
        int count = 0;

        for (int e : nums) {
            if ((e & (1 << bitIndex)) != 0)
                count++;
        }

        if (count % 3 == 1) {
            ans = ans | (1 << bitIndex);
        }
    }
    return ans;
}
```

↳ check if the bit is set

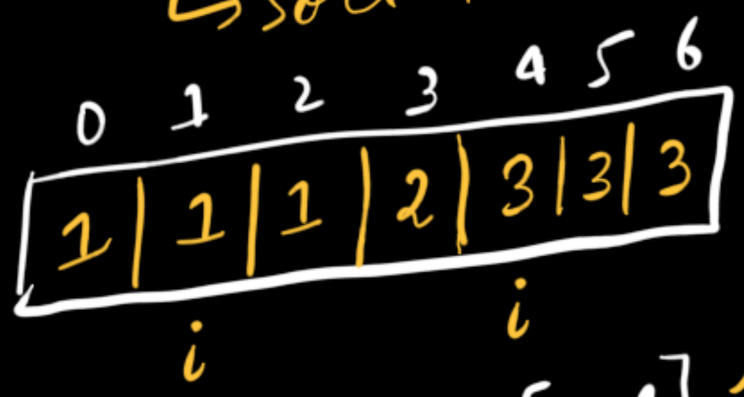
TC: O(N×32)

SC: O(1)

↳ set the bitindex bit

→ Approach-3

↳ Sort the array



- nums[i] != nums[i-1] → i = i+3
  (3 != 2) → return 2
- if no number is returned then last number will be our answer in array.

```java
public static int singleNumberIIBetterII(int[] nums) {
    Arrays.sort(nums);

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

TC: O(n log n) + O(n/3)

SC: O(1)

→ Approach-4

ones = will store bit that have appeared once

twos = will store bit that have appeared twice

↳ Any bit that appear twice will be cleared from both.

→ to get the bit based on e

ones = (ones ^ e) & ~twos;

twos = (twos ^ e) & ~ones;

↳ same for this

↳ ensure part of bit already in twos is cleared

↳ current element

```java
public static int singleNumberOpt(int @NotNull [] nums) {
    int ones = 0;
    int twos = 0;

    for (int e : nums) {
        ones = (e ^ ones) & ~twos;
        twos = (e ^ twos) & ~ones;
    }
    return ones;
}
```

TC: O(N)

SC: O(1)

→ Single Number III

↳ Given an integer array nums, where exactly two elements appear once, & all the other element is appear exactly twice, return the two element that appear only once.

nums = [1|2|1|3|2|5] → [3,5] ↳ answer.

XOR = 1^1^2^2^3^5 => 3^5 = 6 → How do we get 3 & 5 from 6

6 = 0110 → we know that doing XOR will give the bit difference between two numbers.

Two bit that were different in both of the numbers.

→ extract the right set bit

(num & num-1) ^ num

(6 & 5) ^ 6 = 2 (0010)

3: 0011 5: 0101

2: 0010

→ 3 → storing all the 1's bit which are 1 (bucket-1)

→ 5 → all the 1's bit which are 0 (bucket-2)

[1|2|1|3|2|5]

1st_bit

1 = 0001 3 = 0011

2 = 0010 2 = 0010

1 = 0001 5 = 0101

1st bit

bucket-1: 2, 3, 2 → 1st_bit = 1 → 2^2^3 = 3

bucket-2: 5, 1, 1 → 1st_bit = 0 → 1^1^5 = 5

TC: O(2N)

SC: O(1)

PartV.java

```java
public int[] singleNumberIII(int @NotNull [] nums) {
    long xor = 0;

    for (int e : nums) xor = xor ^ e;
    long rightMostSetBit = (xor & (xor - 1)) & xor;

    int b1 = 0;
    int b2 = 0;

    for (int e : nums) {
        if ((e & rightMostSetBit) != 0) b1 = b1 ^ e;
        else b2 = b2 ^ e;
    }

    return new int[]{b1, b2};
}
```

→ XOR of a numbers in a given range

L = 5 , R = 10

↳ ans = 5 ^ 6 ^ 7 ^ 8 ^ 9 ^ 10 = 15

→ Approach-1

```
int XOR=0;
for(int i=L; i<=R; i++){
    XOR = XOR ^ i;
}
```

TC : O(R-L)
SC : O(1)

→ Approach-2

↳ can we find XOR of 1 to N in constant time?

4 step:
- N = 1
- N = 2
- N = 3
- N = 4
- N = 5
- N = 6
- N = 7
- N = 8
- N = 9

1 ^ 2
1 ^ 2 ^ 3
1 ^ 2 ^ 3 ^ 4
1 ^ 2 ^ 3 ^ 4 ^ 5
1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6
1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7
1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7 ^ 8
1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7 ^ 8 ^ 9

1	1
3	0
0	N
4	
1	
7	
0	
8	
1	

	ans
n%4==1	1
n%4==2	N+1
n%4==3	0
n%4==0	N

Now we can find XOR from 1 - N in O(1). How do we find XOR of L to R.

L = 3, R = 5

1 ^ 2 → 0 → (L-1)
1 ^ 2 ^ 3 ^ 4 ^ 5

(L-1) ^ (R)

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

PartVI.java

```java
public static int XorRange(int L , int R){  1 usage
    return findXOROpt( num: L - 1) ^ findXOROpt(R);
}

@Contract(pure = true)
private static int findXOROpt(int num) {  2 usages  &
    if (num % 4 == 0) return num;
    else if (num % 4 == 1) return 1;
    else if (num % 4 == 2) return num + 1;
    else return 0; // num % 4 == 3
}
```

→ Divide two Numbers without / operator

dividend = 22 (a) divisor = 3 (b) so, a/b = 7

→ Approach-1

22/7

1 2 3 4 5 6 7

↳ (3+3+3+3+3+3+3)

21

```
while ( sum + b <= a){
    sum += b;
    count++;
}
```

→ Approach-2

a = 22, b = 3

22
↳ 3×2⁰ = 3
3×2¹ = 6
3×2² = 12 }
3×2³ = 24 ×

22 } 2²

we can reduce 22 by 12

10
↳ 3×2⁰ = 3
3×2¹ = 6 } 2¹
3×2² = 12 ×

4
↳ 3×2⁰ = 3 } 2⁰
3×2¹ = 6 ×

ans = 2² + 2¹ + 2⁰ = 7

Handling -ve numbers → $\frac{-n}{d}$ || $\frac{n}{-d}$: boolean isNegative (dividend < 0 && divisor > 0) || (dividend > 0 && divisor < 0);

```
quotient = 0;
while (n >= d){
    int count = 0;
    while (n >= (d << (count+1))){
        count++;
    }
    quotient += 1 << count;
    n = n - d << count;
}
return isNegative ? -quotient : quotient;
}
```

d × 2^(count+1)

→ 2^count

→ d × 2^count

```
PartVII.java
public static int divide(int dividend, int divisor) {  no usages  bravo1goingdark
    if (dividend == Integer.MIN_VALUE && divisor == -1){
        return Integer.MAX_VALUE;
    }
    if (dividend == divisor) return 1;
    boolean isNegative = dividend < 0 && divisor > 0 || divisor < 0 && dividend > 0;

    int a = Math.abs(dividend);
    int b = Math.abs(divisor);

    int sum = 0;
    int count = 0;

    while (sum + b <= a){
        sum +=.b;
        count++;
    }
    return isNegative ? -count : count;
}
```

TC: O(dividend/divisor)

SC: O(1)

```
PartVII.java
public static int divideOpt(int dividend, int divisor) {  1 usage  bravo1goingdark
    if (dividend == Integer.MIN_VALUE && divisor == -1) {
        return Integer.MAX_VALUE;
    }
    if (dividend == divisor) return 1;

    boolean isNegative = (dividend < 0 && divisor > 0) || (dividend > 0 && divisor < 0);

    long n = Math.abs((long) dividend);
    long d = Math.abs((long) divisor);

    int quotient = 0;

    while (n >= d) {
        int cnt = 0;

        // Double the divisor until it exceeds the remaining dividend
        while (n >= (d << (cnt + 1))) {
            cnt++;
        }

        quotient += 1 << cnt;  // Add 2^cnt times to result
        n -= d << cnt;         // Subtract d * 2^cnt from dividend
    }

    return isNegative ? -quotient : quotient;
}
```

O(log n)

TC: O(log₂ n)

SC: O(1)

→ Sum of two Integers

↳ Add two number without (+, -) operator

a = 12, b = 13

```
  1100
+ 1101
------
 11001
```

a = 31 b = 31 → carry

```
  1 1 1 1 1
    | | | |
   | | | |
+ ---------
  1 1 1 1 1 0
```

^ → do sum without any carry
& → Calculat the carry
<< → carry shifted left

```
a → | | | | |
b → | | | | |
& ---------
  = | | | | |
```

```
a → | | | | |
b → | | | | |
^ ---------
  = 0 0 0 0 0
```

b = c << 1

```
  | | | | | <<1
b = | | | | 0
a = 0 0 0 0 0
```

↳ we will keep doing this process Until b != 0

PartVIII.java

```java
public static int getSum(int a, int b) {  no usages  n
    while (b != 0) {
        int carry = a & b;      // Common set bits
        a = a ^ b;              // Sum without carry
        b = carry << 1;         // Carry shifted left
    }
    return a;
}
```

TC: O(1) → at max run for 32 bits

SC: O(1)