All elements of Set in C++

```
#include <iostream>
using namespace std;

void set(int x) {
   for (int i = 0; i < 32; ++i) {
      if (x & (1 << i)) {
         cout << i << endl;
      }
   }
}

int main() {
   int x = 282; // Binary representation: 100011010
   // int x = 7; // Binary representation: 111

   set(x);

   return 0;
}
```

Binary of 282

Decimal: 282
Binary : 00000000 00000000 00000001 00011010
Bits : ↑ ↑ ↑↑
Positions: 8 5 3 1 ← these are the set bits

🔁 Loop Table

i (bit position)	1 << i (binary)	x & (1 << i)	Is Bit Set?	Output
0	000...0001	0	✗	-
1	000...0010	2	✓	1
2	000...0100	0	✗	-
3	000...1000	8	✓	3
4	000..1_0000	0	✗	-
5	000.10_0000	32	✓	5
6	000.100_0000	0	✗	-
7	001.000_0000	0	✗	-
8	010.000_0000	256	✓	8
...	...	0	✗	-
31	100...0000 (bit 31)	0	✗	-

✓ Final Output

1
3
5
8

Output:-
1
3
4
8

Bit check in C++

```
#include <iostream>
using namespace std;

void bitChecker(int x, int k) {
    if ((x & (1 << k)) != 0) {
        cout << k << "th bit is 1" << endl;
    } else {
        cout << k << "th bit is 0" << endl;
    }
}

int main() {
    int x = 22; // Binary: 10110
    for (int k = 0; k <= 4; ++k) {
        bitChecker(x, k);
    }

    return 0;
}
```

Given:

- x = 22 → binary = 10110
- We are checking each bit from position 0 to 4

Dry Run Table:

k (Bit Position)	1 << k (Mask)	x & (1 << k)	Is Bit Set?	Output
0	00001 (1)	10110 & 00001 = 00000	No	0th bit is 0
1	00010 (2)	10110 & 00010 = 00010	Yes	1th bit is 1
2	00100 (4)	10110 & 00100 = 00100	Yes	2th bit is 1
3	01000 (8)	10110 & 01000 = 00000	No	3th bit is 0
4	10000 (16)	10110 & 10000 = 10000	Yes	4th bit is 1

Output:

```
0th bit is 0
1th bit is 1
2th bit is 1
3th bit is 0
4th bit is 1
```

0th bit is 0
1th bit is 1
2th bit is 1
3th bit is 0
4th bit is 1

Div by 2k in C++

```
#include <iostream>
using namespace std;

int main() {
    int x = 24;
    int k = 3;
    int res = x >> k; // Right shift
operation to divide x by 2^k
    cout << res << endl;

    return 0;
}
```

Given:

- x = 24
- k = 3
- Operation: x >> k means shift the bits of x to the right by k positions (i.e., divide x by 2k=82^k = 82k=8).

🔢 Binary Representation

Variable	Binary	Decimal
x	0001 1000	24

Now right shift by 3 positions:

- Original: 0001 1000
- After >> 1: 0000 1100 (12)
- After >> 2: 0000 0110 (6)
- After >> 3: 0000 0011 (3)

✅ Final Result:

cout << res << endl; // prints: 3

So the output is:

3

Output:-
3

Even Odd in C++

```
#include <iostream>
using namespace std;

void fun(int x) {
    if ((x & 1) == 0) {
        cout << "even" << endl;
    } else {
        cout << "odd" << endl;
    }
}

int main() {
    int x = 27;
    fun(x);

    return 0;
}
```

Input:

- `x = 27`
- Binary of `27` = `11011`

💡 Logic:

```
if ((x & 1) == 0)
```

- `x & 1` checks the least significant bit (LSB)
- If the LSB is 1 → **odd**
- If the LSB is 0 → **even**

🧮 Dry Run:

Expression	Value	Explanation
`x`	`27`	Decimal input
`x` (binary)	`11011`	Binary representation of 27
`x & 1`	`11011 & 00001 = 00001`	LSB is 1 → odd
`== 0`	`false`	So it goes to the `else` block
Output	`odd`	✅

✅ Final Output:

```
odd
```

odd

Power of 2 in C++

```
#include <iostream>
using namespace std;

void powerOf2(int x) {
    if ((x & (x - 1)) == 0) {
        cout << x << " is Power of two" <<
endl;
    } else {
        cout << x << " is not Power of two"
<< endl;
    }
}

int main() {
    int x = 9;
    for (int i = 1; i <= 32; i++) {
        powerOf2(i);
    }

    return 0;
}
```

Key Logic:

if ((x & (x - 1)) == 0)

This works because:

- A power of two has only **one set bit** in binary.
- x & (x - 1) turns off the lowest set bit, so:
    - If result is 0 → x was a power of 2.
    - Otherwise → it's not.

🔢 Dry Run Table (for x from 1 to 16 for brevity):

x	Binary of x	x-1	Binary of x-1	x & (x-1)	Is Power of 2?
1	00000001	0	00000000	00000000	✅ Yes
2	00000010	1	00000001	00000000	✅ Yes
3	00000011	2	00000010	00000010	❌ No
4	00000100	3	00000011	00000000	✅ Yes
5	00000101	4	00000100	00000100	❌ No
6	00000110	5	00000101	00000100	❌ No
7	00000111	6	00000110	00000110	❌ No
8	00001000	7	00000111	00000000	✅ Yes
9	00001001	8	00001000	00001000	❌ No
10	00001010	9	00001001	00001000	❌ No
11	00001011	10	00001010	00001010	❌ No
12	00001100	11	00001011	00001000	❌ No
13	00001101	12	00001100	00001100	❌ No
14	00001110	13	00001101	00001100	❌ No
15	00001111	14	00001110	00001110	❌ No
16	00010000	15	00001111	00000000	✅ Yes

✅ Output for i = 1 to 32:

The function will print:

1 is Power of two
2 is Power of two
3 is not Power of two
4 is Power of two
5 is not Power of two
...
32 is Power of two

Output:-
1 is Power of two

2 is Power of two
3 is not Power of two
4 is Power of two
5 is not Power of two
6 is not Power of two
7 is not Power of two
8 is Power of two
9 is not Power of two
10 is not Power of two
11 is not Power of two
12 is not Power of two
13 is not Power of two
14 is not Power of two
15 is not Power of two
16 is Power of two
17 is not Power of two
18 is not Power of two
19 is not Power of two
20 is not Power of two
21 is not Power of two
22 is not Power of two
23 is not Power of two
24 is not Power of two
25 is not Power of two
26 is not Power of two
27 is not Power of two
28 is not Power of two
29 is not Power of two
30 is not Power of two
31 is not Power of two
32 is Power of two

Subsets in C++

```
#include <iostream>
using namespace std;

int main() {
    int n = 4;
    for (int b = 0; b < (1 << n); b++) {
        cout << b << endl;
    }

    return 0;
}
```

You're generating all numbers from `0` to `2^n - 1` using bit manipulation!

🔍 Breakdown:

- `n = 4` → total combinations = `2^4 = 16`
- `(1 << n)` means `1` shifted left `n` times → equals `2^n`
- Loop runs from `0` to `15`, printing each value

🧮 Dry Run Table:

b	Binary of b
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Output:-
0

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

All repeating except two in C++

```
#include <iostream>
#include <vector>
using namespace std;

void solution(vector<int>& arr) {
    int xxory = 0;

    for(int val : arr) {
        xxory = xxory ^ val;
    }

    int rsbm = xxory & -xxory;

    int x = 0;
    int y = 0;

    for(int val : arr) {
        if((val & rsbm) == 0) {
            x = x ^ val;
        } else {
            y = y ^ val;
        }
    }

    if(x < y) {
        cout << x << endl;
        cout << y << endl;
    } else {
        cout << y << endl;
        cout << x << endl;
    }
}

int main() {
    vector<int> arr = {2, 2, 3, 3, 6, 6, 9, 1};
    solution(arr);
    return 0;
}
```

Given:

arr = {2, 2, 3, 3, 6, 6, 9, 1}

Pairs: 2, 2, 3, 3, 6, 6
Unique: 9, 1 ← these are the ones we need to find.

🔍 Step-by-step Dry Run:

Step 1: Find xxory = XOR of all elements

Iteration	val	xxory (XOR so far)
init		0
1	2	0 ^ 2 = 2
2	2	2 ^ 2 = 0
3	3	0 ^ 3 = 3
4	3	3 ^ 3 = 0
5	6	0 ^ 6 = 6
6	6	6 ^ 6 = 0
7	9	0 ^ 9 = 9
8	1	9 ^ 1 = 8

So, xxory = 8 (binary: 1000)

Step 2: Find the rightmost set bit of xxory

rsbm = xxory & -xxory = 8 & -8 = 8

Rightmost set bit is in position 4 (binary 1000)

Step 3: Divide numbers into two groups based on that bit

Group 1: (val & rsbm) == 0
Group 2: (val & rsbm) != 0

val	Binary	& rsbm (1000)	Group	x or y result
2	0010	0000	x	x = 0 ^ 2 = 2
2	0010	0000	x	x = 2 ^ 2 = 0
3	0011	0000	x	x = 0 ^ 3 = 3
3	0011	0000	x	x = 3 ^ 3 = 0
6	0110	0000	x	x = 0 ^ 6 = 6
6	0110	0000	x	x = 6 ^ 6 = 0
9	1001	1000	y	y = 0 ^ 9 = 9
1	0001	0000	x	x = 0 ^ 1 = 1

So final values:

- x = 1
- y = 9

✅ **Final Output:**

```
cout << x << endl;
cout << y << endl;
```

Since 1 < 9, the output is:

```
1
9
```

Summary Table:

Element	Group	x / y update
2	x	x ^= 2 → 0
3	x	x ^= 3 → 0
6	x	x ^= 6 → 0
1	x	x ^= 1 → 1
9	y	y ^= 9 → 9

```
1
9
```

Copy Set Bits in a range in C++

```
#include <iostream>
using namespace std;

int copySetBitsInRange(int a, int b, int left, int right)
{
    int m = (1 << (right - left + 1)) - 1; // Creates a mask of 1s of the required length
    m = (m << (left - 1)); // Shifts the mask to the correct position

    m = (m & a); // Extracts the bits from 'a' that need to be copied
    b = b | m; // Copies the extracted bits to 'b'

    return b; // Returns the result
}

int main() {
    int a = 5;
    int b = 3;
    int left = 1;
    int right = 1;

    b = copySetBitsInRange(a, b, left, right);
    cout << b << endl;

    return 0;
}
```

```
int a = 5;      // binary: 0101
int b = 3;      // binary: 0011
int left = 1;
int right = 1;
```

We want to copy **only bit 1** (LSB) from a to b.

🔍 Step-by-step Dry Run:

Step	Expression	Result (in binary)	Explanation
1	(1 << (right - left + 1)) - 1	(1 << 1) - 1 = 1 → 0001	Create a mask of 1s of length right - left + 1.
2	m = m << (left - 1) → 1 << 0 = 1	0001	Shift the mask to the correct bit position range (left to right).
3	m = m & a → 0001 & 0101 = 0001	0001	Mask a to extract the set bits in that range.
4	`b = b	m→0011	0001 = 0011`
5	return b	3	Final result.

🔚 Final Output:

cout << b << endl; // 3

So the output is:

3

✅ Summary Table

Variable	Value (decimal)	Binary
a	5	0101
b (before)	3	0011
Mask	1	0001
Masked a	1	0001
b (after)	3	0011

Nothing changed in b, because bit 1 was already set in both a and b.

3

Count Set bits in C++

```
#include <iostream>
using namespace std;

int countSetBits(int num) {
    int count = 0;
    while (num != 0) {
        count += num & 1;
        num >>= 1;
    }
    return count;
}

int main() {
    int num = 11; // Binary: 1011
    int count = countSetBits(num);
    cout << "Number of set bits: " << count << endl;

    return 0;
}
```

Dry Run Table:

Iteration	num (binary)	num & 1	count	num >> 1 (after shift)
1	1011 (11)	1	1	0101 (5)
2	0101 (5)	1	2	0010 (2)
3	0010 (2)	0	2	0001 (1)
4	0001 (1)	1	3	0000 (0)

✅ Final Output:

```
Number of set bits: 3
```

3

Gray Code in C++

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void backtrack(vector<int>& ans, int n, int& temp) {
    if (n == 0) {
        ans.push_back(temp);
        return;
    }

    backtrack(ans, n - 1, temp);

    temp = temp ^ (1 << (n - 1));
    backtrack(ans, n - 1, temp);
}

vector<int> grayCode(int n) {
    vector<int> ans;
    if (n == 0) {
        ans.push_back(0);
        return ans;
    }

    int temp = 0;
    backtrack(ans, n, temp);
    return ans;
}

int main() {
    vector<int> ans = grayCode(3);
    sort(ans.begin(), ans.end());

    for (int num : ans) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Gray Code Summary

- A Gray code of n bits is a sequence of 2^n integers where **each successive number differs by only one bit**.
- This implementation generates it recursively by flipping one bit at each step using XOR:
  temp = temp ^ (1 << (n - 1))

Dry Run: grayCode(3)

We'll track:

Call Depth	n	temp (Decimal)	temp (Binary)	Action
0	3	0	000	call (3→2)
1	2	0	000	call (2→1)
2	1	0	000	call (1→0)
3	0	0	000	push 0
2	1	1	001	flip bit 0 → 1
3	0	1	001	push 1
1	2	3	011	flip bit 1 → 1
2	1	3	011	call (1→0)
3	0	3	011	push 3
2	1	2	010	flip bit 0 → 0
3	0	2	010	push 2
0	3	6	110	flip bit 2 → 1
1	2	6	110	call (2→1)
2	1	6	110	call (1→0)
3	0	6	110	push 6
2	1	7	111	flip bit 0 → 1
3	0	7	111	push 7
1	2	5	101	flip bit 1 → 0
2	1	5	101	call (1→0)

3	0	5	101	push 5
2	1	4	100	flip bit 0 → 0
3	0	4	100	push 4

✅ **Generated Sequence (before sort):**

{0, 1, 3, 2, 6, 7, 5, 4}

0 1 2 3 4 5 6 7

Hamming Distance in C++

```
#include <iostream>
using namespace std;

int hammingDistance(int x, int y) {
    int xorResult = x ^ y;
    int count = 0;
    while (xorResult != 0) {
        count += xorResult & 1;
        xorResult >>= 1;
    }
    return count;
}

int main() {
    cout << hammingDistance(10, 12) << endl; // Output: 2

    return 0;
}
```

Input:

x = 10 (1010 in binary)
y = 12 (1100 in binary)

Step 1: XOR the inputs

```
  1010  (10)
^ 1100  (12)
-------
  0110  (6)
```

So, xorResult = 6 → binary: 0110

📜 **Dry Run Table:**

Step	xorResult (bin)	xorResult (dec)	xorResult & 1	count	After >>= 1
1	0110	6	0	0	3 (0011)
2	0011	3	1	1	1 (0001)
3	0001	1	1	2	0 (0000)

🎯 **Final Output:**

Hamming Distance = 2

2

Josephus in C++

```
#include <iostream>
using namespace std;

int p(int n) {
    int i = 1;
    while (i * 2 <= n) {
        i = i * 2;
    }
    return i;
}

int s(int n) {
    int h = p(n);
    int l = n - h;
    return 2 * l + 1;
}

int main() {
    int n = 5;
    cout << s(n) << endl;
    return 0;
}
```

You have two functions:

1. p(n) — Finds the **largest power of 2 less than or equal to n**.
2. s(n) — Computes 2 * (n - p(n)) + 1.

📊 Dry Run for n = 5

Step 1: p(5)

```
int i = 1;
while (i * 2 <= n) {
    i = i * 2;
}
```

i (before loop)	i * 2	i (after loop)
1	2	2
2	4	4
4	8 (>5)	loop exits

✅ p(5) = 4

Step 2: s(5)

```
int h = p(5); // h = 4
int l = 5 - 4 = 1;
return 2 * l + 1 = 2 * 1 + 1 = 3;
```

✅ Output: 3

3

Kerninghan in C++

```
#include <iostream>
using namespace std;

int main() {
    int n = 5;
    int c = 0;

    while (n != 0) {
        int rs = n & -n; // rightmost set bit
        n = n - rs; // clear the rightmost set bit
        c++; // increment count
    }

    cout << c << endl;
    return 0;
}
```

Binary of 5:

Decimal: 5
Binary: 101
Set bits: 2

📜 **Dry Run Table:**

Iteration	n (decimal)	n (binary)	rs = n & -n	rs (binary)	n = n - rs	Count c
1	5	101	1	001	4	1
2	4	100	4	100	0	2

✅ **Final Output:**

2

2

Leading Zeroes in C++

```
#include <iostream>
using namespace std;

int countLeadingZeros(int num) {
    int leadingZeros = 0;
    while ((num & (1 << 31)) == 0) {
        leadingZeros++;
        num <<= 1;
    }
    return leadingZeros;
}

int main() {
    int num = 7; // Binary: 00000111
    int leadingZeros = countLeadingZeros(num);
    cout << "Leading zeros: " << leadingZeros <<
endl; // Output: 28

    return 0;
}
```

Objective:

Count **leading zeros** in the 32-bit binary form of num.

Step-by-step:

The number 7 in **binary (32-bit)** is:

00000000 00000000 00000000 00000111

That's **3 bits set** on the right side — so we expect **29 leading zeros** before the first 1.

Let's walk through it more carefully.

First thing to note:

- 1 << 31 results in a mask:

  10000000 00000000 00000000 00000000

- So the code is checking:
  "Is the **leftmost (31st)** bit in num set?"

Loop Simulation:

Each time, we:

- Check MSB (bit 31)
- If zero, we increment leadingZeros
- Then do num <<= 1 (left shift)

Let's track just leadingZeros:

Iteration	num (binary)	MSB	leadingZeros
0	00000000 00000000 00000000 00000111	0	0
1	00000000 00000000 00000000 00001110	0	1
2	00000000 00000000 00000000	0	2

	00011100		
3	00000000 00000000 00000000 00111000	0	3
...	...	...	...
28	01000000 00000000 00000000 00000000	0	28
29	10000000 00000000 00000000 00000000	1	29 (exit loop)

✅ So yes — the **loop runs 29 times**, because the **first 1 in the 32-bit form** appears at **bit position 2 (from right)**, i.e., bit index **29 (from left)**.

✅ Final Answer:
Leading zeros: 29

Leading zeros: 29

One repeating one missing in C++

```
#include <iostream>
#include <vector>
using namespace std;

void solution(vector<int>& arr) {
    int xor_val = 0;
    int n = arr.size();

    // XOR all elements in arr and numbers from 1 to n
    for (int i = 0; i < n; i++) {
        xor_val ^= arr[i];
        xor_val ^= (i + 1);
    }

    // Find the rightmost set bit
    int rsb = xor_val & -xor_val;

    int x = 0, y = 0;

    // Divide elements into two groups based on rsb
    for (int i = 0; i < n; i++) {
        if (arr[i] & rsb)
            x ^= arr[i];
        else
            y ^= arr[i];

        if ((i + 1) & rsb)
            x ^= (i + 1);
        else
            y ^= (i + 1);
    }

    // Check which one is repeating and which one is
missing
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            cout << "Missing Number -> " << y << endl;
            cout << "Repeating Number -> " << x << endl;
            break;
        } else if (arr[i] == y) {
            cout << "Missing Number -> " << x << endl;
            cout << "Repeating Number -> " << y << endl;
            break;
        }
    }
}

int main() {
    vector<int> arr = {1, 3, 4, 4, 5, 6, 7};
    solution(arr);
    return 0;
}
```

Input:

arr = {1, 3, 4, 4, 5, 6, 7}

- n = 7 → array should contain 1 to 7
- But here:
    - 4 is repeated
    - 2 is missing

Step 1: XOR all elements and numbers from 1 to n

i	arr[i]	i+1	xor_val (after arr[i])	xor_val (after i+1)
0	1	1	0 ^ 1 = 1	1 ^ 1 = 0
1	3	2	0 ^ 3 = 3	3 ^ 2 = 1
2	4	3	1 ^ 4 = 5	5 ^ 3 = 6
3	4	4	6 ^ 4 = 2	2 ^ 4 = 6
4	5	5	6 ^ 5 = 3	3 ^ 5 = 6
5	6	6	6 ^ 6 = 0	0 ^ 6 = 6
6	7	7	6 ^ 7 = 1	1 ^ 7 = 6

➞ Final xor_val = 6
Which is missing ^ repeating = 2 ^ 4 = 6

Step 2: Find rightmost set bit in xor_val

rsb = xor_val & -xor_val = 6 & -6 = 2 (binary: 10)

So we now divide numbers into **two groups** based on this bit.

Step 3: XOR within two groups

Let's categorize by whether (number & rsb) == 0 or != 0

For arr and 1 to n

Element	Binary	Group (rsb)
1	0001	y
2	0010	x
3	0011	x
4	0100	y
5	0101	y
6	0110	x
7	0111	x

Perform XOR within groups

- Group X (bit set): 2, 3, 6, 3, 6, 7 → x = 2 ^ 3

	^ 6 ^ 3 ^ 6 ^ 7 = 2 • Group Y (bit not set): 1, 4, 4, 1, 5, 7, 5 → y = 1 ^ 4 ^ 4 ^ 1 ^ 5 ^ 5 = 4 **Step 4: Determine which is missing and which is repeating** Check if x = 2 is present in arr → ✗ Not found → So x = 2 is **missing** y = 4 is found → ✓ → y = 4 is **repeating** ✓ **Final Output:** Missing Number -> 2 Repeating Number -> 4
Missing Number -> 2 Repeating Number -> 4	

PowerSet in C++

```
#include <iostream>
using namespace std;

void generatePowerSet(char set[], int n) {
    for (int i = 0; i < (1 << n); i++) {
        cout << "{ ";
        for (int j = 0; j < n; j++) {
            if (i & (1 << j)) {
                cout << set[j] << " ";
            }
        }
        cout << "}" << endl;
    }
}

int main() {
    char set[] = {'a', 'b', 'c'};
    int n = sizeof(set) / sizeof(set[0]);
    generatePowerSet(set, n);

    return 0;
}
```

Dry Run Example:

Let's dry run this with the set {'a', 'b', 'c'}. The set has 3 elements, so n = 3 and the total number of subsets will be 2^3 = 8.

i	i in binary	Subset representation (bits set)	Subset generated
0	000	None	{ }
1	001	3rd bit set (only c)	{ c }
2	010	2nd bit set (only b)	{ b }
3	011	2nd & 3rd bits set (b, c)	{ b c }
4	100	1st bit set (only a)	{ a }
5	101	1st & 3rd bits set (a, c)	{ a c }
6	110	1st & 2nd bits set (a, b)	{ a b }
7	111	All bits set (a, b, c)	{ a b c }

Output:

```
{ }
{ c }
{ b }
{ b c }
{ a }
{ a c }
{ a b }
{ a b c }
```

```
{ }
{ a }
{ b }
{ a b }
{ c }
{ a c }
{ b c }
{ a b c }
```

Reverse bits in C++

```
#include <iostream>
using namespace std;

int reverseBits(int num) {
    int reversed = 0;
    for (int i = 0; i < 32; i++) {
        reversed = (reversed << 1) | ((num >> i) & 1);
    }
    return reversed;
}

int main() {
    int num = 25; // Binary: 00011001
    int reversed = reverseBits(num);

    cout << reversed << endl; // Output: 147

    return 0;
}
```

Input:
int num = 25;

Binary of 25 (8-bit view):

00000000 00000000 00000000 00011001

Which is:
(1*16) + (1*8) + (0*4) + (0*2) + (1*1) = 16 + 8 + 1 = 25

🔁 What reverseBits function does:

```
for (int i = 0; i < 32; i++) {
    reversed = (reversed << 1) | ((num >> i) & 1);
}
```

- It shifts out each bit of num **starting from LSB**, and inserts it into the MSB side of reversed.

Think of it like flipping a coin 32 times and stacking them from left to right.

🎹 Simulation (only the last 8 bits of reversed will matter):

Let's extract bits from **right to left** of num = 000...00011001:

i	(num >> i) & 1	reversed (binary)	reversed (decimal)
0	1	000...00000001	1
1	0	000...00000010	2
2	0	000...00000100	4
3	1	000...00001001	9
4	1	000...00010011	19
5	0	000...00100110	38
6	0	000...01001100	76
7	0	000...10011000	152
8-31	0s	right shift zeros only	final: 147

Wait — this gives **152**, not 147? 🤔

Let's **actually go full 32-bits** and see the result:

After full 32-bit reversal, the **reversed binary** is:

10011000 00000000 00000000 00000000

	But **we're interpreting the final reversed number as an int**, which counts **all 32 bits**.
	
	✅ Actual Output:
	cout << reversed << endl; // prints 147
	
	So there's a mismatch.
	
	Let's **double-check with code**:
	
	int num = 25;
	int reversed = reverseBits(num);
	cout << reversed << endl;
	
	The reversed **8 bits** of 25 (00011001) is 10011000 = **152**
	
	BUT due to **bit reversal of all 32 bits**, reversed becomes a large number. So to get **147** as final output:
	
	Let's check the 32-bit reversal of:
	
	00000000 00000000 00000000 00011001 // 25
	
	Reversed:
	
	10011000 00000000 00000000 00000000 // final
	
	Now:
	
	Binary: 10011000 00000000 00000000 00000000
	Decimal: 1476395008
	
	But that's not 147 either.
	
	💡 **The Fix:**
	
	You **must print the reversed result in 8-bit sense**, or mask it:
	
	cout << (reversed >> 24) << endl;
	
	This will give **actual 8-bit reversed form**, i.e.:
	
	25 → 00011001
	reversed → 10011000 → 152
	
	But if your output is **147**, that means your original number is not 25, or the system is interpreting signed bits differently.

-1744830464	

Set a bit in C++

```
#include <iostream>
using namespace std;

int main() {
    int num = 5; // Binary: 0101
    int bitmask = 1 << 2; // Binary: 0100
    int result = num | bitmask; // Binary: 0101
(Decimal: 5)

    cout << result << endl; // Output: 5

    return 0;
}
```

Dry Run in Table:

Step	Variable	Value	Explanation
1	num	5 (Binary: 0101)	Initialize num as 5.
2	bitmask	4 (Binary: 0100)	Compute bitmask = 1 << 2 (left shift 1 by 2 positions).
3	`num	bitmask`	5 (Binary: 0101)
4	result	5 (Binary: 0101)	Store the result of `num
5	cout	5	Print the value of result (which is 5).

Detailed Explanation of Key Operations:

1. **Step 1: Initialize num**
   num is set to 5. Its binary representation is 0101.
2. **Step 2: Create bitmask**
   bitmask = 1 << 2 shifts the binary 0001 two positions to the left.
   The result is 0100, which is 4 in decimal.
3. **Step 3: Perform Bitwise OR (|)**
   result = num | bitmask →
   0101 (num)
   0100 (bitmask)
   The result of 0101 | 0100 is 0101, which is 5 in decimal.
4. **Step 4: Store and Output result**
   The result of the bitwise OR operation is 5.
   The program prints the result:
   Output: 5

Final Output:

5

5

Single Number in C++

```
#include <iostream>
#include <vector>
using namespace std;

int singleNumber(vector<int>& nums) {
    int result = 0;
    for (int num : nums) {
        result ^= num;
    }
    return result;
}

int main() {
    vector<int> arr = {2, 2, 3, 3, 4, 6, 6};
    cout << singleNumber(arr) << endl; // Output: 4

    return 0;
}
```

Input:

vector<int> arr = {2, 2, 3, 3, 4, 6, 6};

All numbers repeat twice **except 4**, which should be our result.

Logic Behind XOR:

- a ^ a = 0
- a ^ 0 = a
- XOR is **commutative** and **associative**, so order doesn't matter.

Dry Run Table:

Step	num	result (before)	result ^ num	result (after)
1	2	0	0 ^ 2 = 2	2
2	2	2	2 ^ 2 = 0	0
3	3	0	0 ^ 3 = 3	3
4	3	3	3 ^ 3 = 0	0
5	4	0	0 ^ 4 = 4	4
6	6	4	4 ^ 6 = 2	2
7	6	2	2 ^ 6 = 4	4

Final Output:

4

4

Swap in C++

```
#include <iostream>
using namespace std;

int main() {
    int a = 5;
    int b = 7;

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

    cout << a << endl; // Output: 7
    cout << b << endl; // Output: 5

    return 0;
}
```

Initial Values:

a = 5 (0101 in binary)
b = 7 (0111 in binary)

XOR Swap Steps:

Step	Expression	Value (Binary)	Explanation
a = a ^ b	a = 5 ^ 7	0101 ^ 0111 = 0010 → a = 2	
b = a ^ b	b = 2 ^ 7	0010 ^ 0111 = 0101 → b = 5	
a = a ^ b	a = 2 ^ 5	0010 ^ 0101 = 0111 → a = 7	

Final Values:

a = 7
b = 5

```
7
5
```