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 \ll x \ll endl;
     cout \ll y \ll endl;
  } else {
     cout \ll y \ll endl;
     cout \ll x \ll 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, $1 \leftarrow$ these are the ones we need to find.

Q 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)

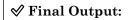
Step 3: Divide numbers into two groups based on that bit

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	у	$y = 0 ^9 = 9$
1	0001	0000	x	$x = 0 ^ 1 = 1$

So final values:

- \bullet x = 1
- y = 9



cout << x << endl; cout << y << endl;</pre>

Since 1 < 9, the output is:

1

Summary Table:

Element	Group	x/y update
2	X	$x \stackrel{\wedge}{=} 2 \rightarrow 0$
3	x	$x \stackrel{\wedge}{=} 3 \rightarrow 0$
6	X	$x \stackrel{\wedge}{=} 6 \rightarrow 0$
1	x	x ^= 1 → 1
9	У	$y = 9 \rightarrow 9$

```
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 \le (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;
```

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

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

Q Step-by-step Dry Run:

Step	Expression	Result (in binary)	Explanation
1	(1 << (right - left + 1)) - 1	$(1 << 1) - 1 = 1 \rightarrow 0001$	Create a mask of 1s of length right - left + 1.
2	$m = m << (left - 1) \rightarrow 1 << 0 = 1$	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:

∜ 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.

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;
}</pre>
```

Dry Run Table:

Iteration	num (binary)	num &	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

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 2ⁿ 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))

III 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 $2 \rightarrow 1$
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 $1 \rightarrow 0$
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
∜ Gei	nerate	ed Seque	ence (before s	sort):
{0, 1, 3	3, 2, 6,	7, 5, 4}		

#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

Hamming Distance in C++

Input:

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

Step 1: XOR the inputs

```
1010 (10)

^ 1100 (12)

-----

0110 (6)
```

So, xorResult = $6 \rightarrow \text{binary: } 0110$

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

return 0;

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;
  \text{return 2 * l + 1};\\
}
int main() {
  int n = 5;
  cout \ll s(n) \ll 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.

 \blacksquare 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

ewline 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

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;
}</pre>
```

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
1	5	101	1	001	4	1
2	4	100	4	100	0	2

∜ Final Output:

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;
}</pre>
```

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?"

Hoop 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	0	0
1	00000111 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	time	s, because the

 \forall 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)**.

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];
       y = arr[i];
    if ((i + 1) \& rsb)
       x = (i + 1);
    else
       y = (i + 1);
  // Check which one is repeating and which one is
  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:

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

- $n = 7 \rightarrow \text{array should contain } 1 \text{ to } 7$
- But here:
 - o 4 is repeated
 - o 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

 \rightarrow 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

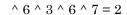
Let's categorize by whether (number & rsb) == 0 or != 0

For arr and 1 to n

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

Perform XOR within groups

• Group X (bit set): 2, 3, 6, 3, 6, $7 \rightarrow x = 2 ^3$



• Group Y (bit not set): 1, 4, 4, 1, 5, 7, 5 \rightarrow 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 $\rightarrow X$ Not found \rightarrow So x = 2 is **missing** y = 4 is found $\rightarrow \emptyset \rightarrow y = 4$ 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;
}</pre>
```

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}
{bc}
{a}
{ac}
{ab}
{abc}
```

```
{}
{a}
{a}
{b}
{ab}
{c}
{ac}
{ac}
{bc}
{ab}
```

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;
}</pre>
```

```
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.

M 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	00000000001	1
1	0	00000000010	2
2	0	00000000100	4
3	1	00000001001	9
4	1	00000010011	19
5	0	00000100110	38
6	0	00001001100	76
7	0	00010011000	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.

So there's a mismatch.

Let's double-check with code:

int num = 25; int reversed = reverseBits(num); cout << reversed << endl;</pre>

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.:

 $25 \rightarrow 00011001$ reversed $\rightarrow 10011000 \rightarrow 152$

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;
}</pre>
```

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:

 ${\bf Output:}\ 5$

Final Output:

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;
}</pre>
```

Input:

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

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

P Logic Behind XOR:

- $a \wedge a = 0$
- $a \land 0 = a$
- XOR is **commutative** and **associative**, so order doesn't matter.

Table: 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

```
#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;
}
```

Swap in C++

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	$0101 \land 0111 = 0010 \rightarrow a = 2$	
b = a ^ b	b = 2 ^ 7	$0010 \land 0111 = 0101 \rightarrow b = 5$	
a = a ^ b	a = 2 ^ 5	$0010 \land 0101 = 0111 \rightarrow a = 7$	

∜ Final Values:

$$a = 7$$

 $b = 5$