Bit Manipulation

- Shivansh

Goal

- Learn what a number base is.
- Understand bit manipulation and its use cases.
- Learn tips for bit manipulation.
- Learn bitwise operators and their properties.
- Learn some common bitwise formulas.
- Learn brute-forcing using bitmasks.

Number Base

A number base is the number of unique digits that are used to represent numbers.

For example:

- Base 10 (decimal): 123, 4141, 9999999
- Base 2 (binary): 1111011, 1000000101101, 100110001001011001111111
- Base 16 (hexadecimal): 7b, 102d, 98967f

When there are not enough digits, alphabets are used instead.

Number Base

To read a number “S” in base “N”, we can use the following formula:

$$\sum_{i=0}^{|S|-1} S_i \times N^i$$

What is Bit Manipulation?

Anything that involves using the digits (bits) of the binary form of a number is bit manipulation.

Usage of operators such as NOT, OR, AND, XOR, LSHIFT, RSHIFT, etc. count as bit manipulation.

We need to figure out how to utilize the above operators with our algorithm to solve a problems.

Where is Bit Manipulation used?

Problems may or may not directly involve bit manipulation.

Some problems use bitwise operators in the problem statement itself, whereas some problems might use bit manipulation indirectly.

For example, if the problem involves powers of 2, it might be related to binary. Similarly, powers of other numbers might be in their own base.

How to approach Bit Manipulation problems?

“When solving a bitwise problem, think in bitwise”

- My mentor, 2020

When solving a problem that uses addition, multiplication, etc. we use decimal, which is the appropriate base for us.

When solving a problem that uses a bitwise operator, we need to use binary, which is the appropriate base.

Miscellaneous

- To print a binary number, we can use

```
cout << bitset<size>(n) << endl;
```

- Make sure “size” is not too large.

- The following bitwise operators are the fastest operators in C++. They are faster than even basic arithmetic operators.

Bitwise Operators - NOT

The NOT operator ("~") flips every bit of the given number.
For example: $\sim 12 = -13$

A	Result
1	0
0	1

Bitwise Operators - OR

The OR operator ("|") takes every corresponding bit of the two numbers and checks whether ***at least one*** of them is set.

A	B	Result
1	1	1
1	0	1
0	1	1
0	0	0

Bitwise Operators - AND

The AND operator (“&”) takes every corresponding bit of the two numbers and checks whether ***both*** of them is set.

A	B	Result
1	1	1
1	0	0
0	1	0
0	0	0

Bitwise Operators - XOR

The XOR operator (“^”) takes every corresponding bit of the two numbers and checks whether ***exactly one*** of them is set.

A	B	Result
1	1	0
1	0	1
0	1	1
0	0	0

Bitwise Operators - Misc

- Bitwise Left Shift (“<<”):
  $A \ll B$ shifts the A to the left by B bits, adding B zeros at the end.
  This value is the same as $A \times 2^B$.

$$18 \ll 3 = 144$$

- Bitwise Right Shift (“>>”):
  $A \gg B$ shifts the A to the right by B bits, deleting B zeros from the end.
  This value is the same as floor of $A \div 2^B$.

$$18 \gg 3 = 2$$

Basic Properties

- Every bitwise operator is associative and commutative.
- $A \wedge 0 = A$
- $A \wedge A = 0$
- If $A \wedge B = C$, then $A \wedge C = B$
- $A \wedge B \wedge B = A$
- $A \,\&\, B \leq \min(A, B)$
- $A \mid B \geq \max(A, B)$
- $(A \mid B) + (A \,\&\, B) = A + B$

Problem 1

Given an array **arr** having **n** positive integers **A_1, A_2, ..., A_n**, followed by

some queries of type **[L , R]** where **queries[i] = [L_i , R_i]** ,

for each query i compute the **XOR** of elements from **L_i** to **R_i** (that is, **arr[L_i] ^ arr[L_{i+1}] ^ ... ^ arr[R_i]**).

Return an array containing the result for the given queries.

Constraints: **$1 <= n <= 10^5$, $1 <= q <= 10^5$, $1 <= L_i <= R_i <= n$**

Problem Link : Here

Problem 2

Benjamin is very fond of finding new tricks for converting Binary to Decimal numbers and vice versa.

He initially had an array **arr** having **n** positive integers $A_1, A_2, ..., A_n$ in his mind, but due to security issues, he does not want to reveal the sequence to you.

But he is ready to respond to no more than 2^***n** of the following questions:

- The Result between the Bitwise AND of two items with indices i and j ($i \neq j$).
- The Result between the Bitwise OR of two items with indices i and j ($i \neq j$).

Can you find the K_{th} largest element of the initial sequence which Benjamin has in his mind.

Constraints: $3 <= n <= 10^4$, $1 \leq K \leq n$

More on Bitwise Operators

More properties:

https://stackoverflow.com/questions/12764670/are-there-any-bitwise-operator-laws

Useful tricks and formulas:

- https://www.geeksforgeeks.org/bitwise-hacks-for-competitive-programming/
- https://www.geeksforgeeks.org/bit-tricks-competitive-programming/
- https://www.geeksforgeeks.org/bits-manipulation-important-tactics/
- https://www.geeksforgeeks.org/builtin-functions-gcc-compiler/

Bitmasking

A bitmask is a sequence of N bits that encodes a subset, where the element is taken if a bit is set, and not taken if a bit is unset.

For example, 10110 would mean $1, 2, 4$ are taken, while $0, 3$ are not.

By generating all bitmasks of some size, we can easily generate all subsets of an array.

No.	Bit representation	Set
0	[0 0 0]	{ }
1	[0 0 1]	{ 1 }
2	[0 1 0]	{ 2 }
3	[0 1 1]	{ 1 , 2 }
4	[1 0 0]	{ 3 }
5	[1 0 1]	{ 1 , 3 }
6	[1 1 0]	{ 2 , 3 }
7	[1 1 1]	{ 1 , 2 , 3 }

<--- No Set Bit

Bitmasking – Brute Force Code

```
for (int mask = 0; mask < (1 << n); mask++)
{
    for (int i = 0; i < n; i++)
        if ((mask >> i) & 1)
            cout << a[i] << " ";

    cout << endl;
}
```

Problems

- https://www.codechef.com/submit/XORMUL

Resources

Number base:

https://brilliant.org/wiki/number-base/

Bit manipulation:

- https://codeforces.com/blog/entry/73490 (highly recommended)
- https://www.hackerearth.com/practice/basic-programming/bit-manipulation/basics-of-bit-manipulation/tutorial/

Bitsets:

- https://www.youtube.com/watch?v=jqJ5s077OKo

Resources

On bitwise operators:

https://medium.com/biffures/bits-101-120f75aeb75a

https://medium.com/biffures/part-2-the-beauty-of-bitwise-and-or-cdf1d8d87891

https://medium.com/biffures/part-3-or-and-20ccc9938f05

https://medium.com/biffures/part-4-bitwise-patterns-7b17dae3eee0