TOPIC - MATRIX AND BIT MANIPULATION

MATRIX

Multidimensional Arrays

We can define multidimensional arrays in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).

General form of declaring N-dimensional arrays:

```
data_type  array_name[size1][size2]....[sizeN];

data_type: Type of data to be stored in the array.

           Here data_type is valid C/C++ data type

array_name: Name of the array

size1, size2,... ,sizeN: Sizes of the dimensions
```

Examples:

Two dimensional array: int two_d[10][20];

Three dimensional array: int three_d[10][20][30];

Size of multidimensional arrays

Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

For example:

The array **int x[10][20]** can store total (10*20) = 200 elements.

Similarly array **int x[5][10][20]** can store total (5*10*20) = 1000 elements.

Two-Dimensional Array

Two – dimensional array is the simplest form of a multidimensional array. We can see a two – dimensional array as an array of one – dimensional array for easier understanding.

- The basic form of declaring a two-dimensional array of size x, y:

  Syntax: data_type array_name[x][y];

  data_type: Type of data to be stored. Valid C/C++ data type.

- We can declare a two dimensional integer array say 'x' of size 10,20 as:

  int x[10][20];

- Elements in two-dimensional arrays are commonly referred by x[i][j] where i is the row number and 'j' is the column number.
- A two – dimensional array can be seen as a table with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and column number ranges from 0 to (y-1). A two – dimensional array 'x' with 3 rows and 3 columns is shown below:

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Initializing Two – Dimensional Arrays:

There are two ways in which a Two-Dimensional array can be initialized.

First Method:

int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}

The above array has 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be filled in the array in the order, first 4 elements from the left in first row, next 4 elements in second row and so on.

Better Method:

int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};

This type of initialization makes use of nested braces. Each set of inner braces represents one row. In the above example there are a total of three rows so there are three sets of inner braces.

Accessing Elements of Two-Dimensional Arrays:

Elements in Two-Dimensional arrays are accessed using the row indexes and column indexes.

Example:

int x[2][1];

The above example represents the element present in the third row and second column.

Note: In arrays if size of array is N, its index will be from 0 to N-1. Therefore, for row index 2 row number is 2+1 = 3.

To output all the elements of a Two-Dimensional array we can use nested for loops. We will require two for loops. One to traverse the rows and another to traverse columns.

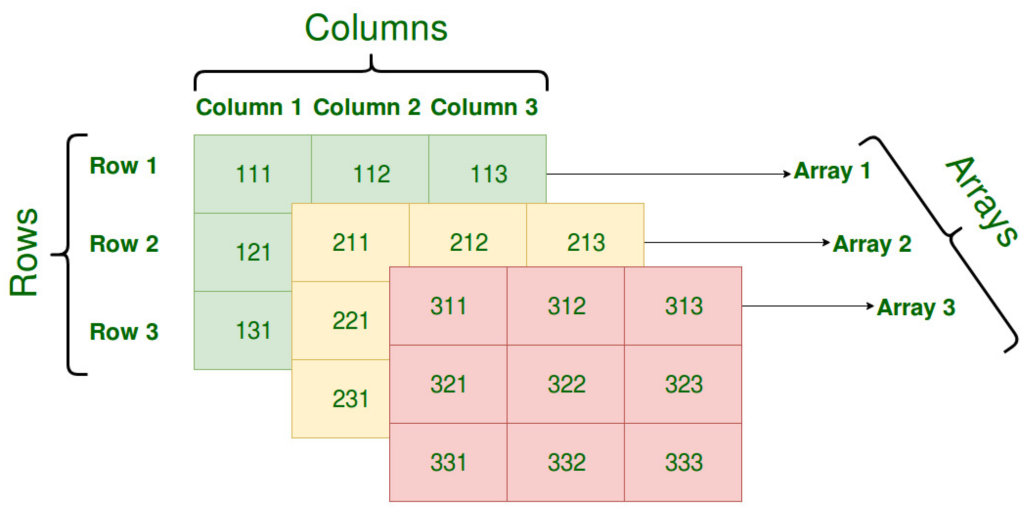
```
#include<iostream>
using namespace std;

int main()
{
     // an array with 3 rows and 2 columns.
     int x[3][2] = {{0,1}, {2,3}, {4,5}};

     // output each array element's value
     for (int i = 0; i < 3; i++)
     {
          for (int j = 0; j < 2; j++)
          {
               cout << x[i][j]<<" ";
          }
          cout<<endl;
     }

     return 0;
}
```

Three-Dimensional Array



Initializing Three-Dimensional Array:

Initialization in a Three-Dimensional array is the same as that of Two-dimensional arrays. The difference is as the number of dimensions increases so the number of nested braces will also increase.

Method 1:

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                  11, 12, 13, 14, 15, 16, 17, 18,
                  19, 20, 21, 22, 23};
```

Better Method:

```
int x[2][3][4] =
{
  { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
  { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }
};
```

Accessing elements in Three-Dimensional Arrays:

Accessing elements in Three-Dimensional Arrays is also similar to that of Two-Dimensional Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in Three-dimensional Arrays.

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

int main() {
     // initializing the 3-dimensional array
     int x[2][3][2] = {
          { {0,1}, {2,3}, {4,5} },
          { {6,7}, {8,9}, {10,11} }
     };

     // output each element's value
     for (int i = 0; i < 2; ++i) {
          for (int j = 0; j < 3; ++j) {
               for (int k = 0; k < 2; ++k) {
                    cout << x[i][j][k] << " " << endl;
               }
          }
     }
     return 0;
}
```

In similar ways, we can create arrays with any number of dimensions. However the complexity also increases as the number of dimensions increases. The most used multidimensional array is the Two-Dimensional Array.

Matrices as Vector of Vectors:

Vector of Vectors is a two-dimensional vector with a variable number of rows where each row is vector. Each index of a vector stores a vector which can be traversed and accessed using iterators. It is similar to an Array of Vectors but with dynamic properties.

Syntax:

vector<vector<data_type>> vec;

Example:

```
vector<vector<int>> vec{ { 1, 2, 3 },
                         { 4, 5, 6 },
                         { 7, 8, 9, 4 } };
```

where vec is the vector of vectors with different number of elements in different rows.

Insertion in Vector of Vectors

Elements can be inserted into a vector using the push_back() function of C++ STL.

Below example demonstrates the insertion operation in a vector of vectors. The code creates a 2D vector by using the push_back() function and then displays the matrix.

Syntax:

```
vector_name.push_back(value)
```

where *value* refers to the element to be added in the back of the vector

Example 1:

v2 = {1, 2, 3}

v1.push_back(v2);

This function pushes vector v2 into the vector of vectors v1. Therefore v1 becomes { {1, 2, 3} }.

Removal or Deletion in a Vector of Vectors

Elements can be removed from a vector of vectors using the [pop_back()]() function of C++ STL.

Below example demonstrates the removal operation in a vector of vectors. The code removes elements from a 2D vector by using the pop_back() function and then displays the matrix.

Syntax:

```
vector_name[row_position].pop_back()
```

Example 1: Let the vector of vectors be vector v = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }}

v[2].pop_back()

This function removes element 9 from the last row vector. Therefore v becomes { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8 } }.

Example 2:

v[1].pop_back()

This function removes element 6 from the last second row vector. Therefore v becomes { { 1, 2, 3 }, { 4, 5 }, { 7, 8 } }.

Traversal of a Vector of Vectors:

The vector of vectors can be traversed using the iterators in C++. The following code demonstrates the traversal of a 2D vector.

Syntax:

```
for i in [0, n)
{
    for (iterator it = v[i].begin();
         it != v[i].end(); it++)
   {
        // Operations to be done
        // For example to print
        print(*it)
    }
}
```

Array of Vectors:

Array of Vectors is a two dimensional array with a fixed number of rows where each row is a vector of variable length. Each index of the array stores a vector which can be traversed and accessed using iterators.

Syntax:

```
vector <data_type> V[size];
```

Example:

vector <int> A[5];

where A is the array of vectors of int of size 5

Insertion: Insertion in an array of vectors is done using push_back() function.

For Example:

```
for i in [0, n) {
  A[i].push_back(35)
}
```

Above pseudo-code inserts element 35 at every index of vector <int> A[n].

Traversal: Traversal in an array of vectors is performed using iterators.

For Example:

```
for i in [0, n) {
   for(iterator it = A[i].begin();
       it!=A[i].end(); it++) {
      print(*it)
    }
}
```

Above pseudo-code traverses vector <int> A[n] at each index using starting iterators A[i].begin() and ending iterator A[i].end(). For accessing the element it uses (*it) as iterators are pointers pointing to elements in vector <int> A[n].

MATRIX

A matrix represents a collection of numbers arranged in an order of rows and columns. It is necessary to enclose the elements of a matrix in parentheses or brackets.

A matrix with 9 elements is shown below.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

This Matrix [M] has 3 rows and 3 columns. Each element of matrix [M] can be referred to by its row and column number. For example, $a_{23}=6$

Order of a Matrix :

The order of a matrix is defined in terms of its number of rows and columns.

Order of a matrix = No. of rows ×No. of columns

Therefore Matrix [M] is a matrix of order 3 × 3.

Transpose of a Matrix :

The transpose $[M]^T$ of an m x n matrix [M] is the n x m matrix obtained by interchanging the rows and columns of [M].

if A= $[a_{ij}]$ mxn , then AT = $[b_{ij}]$ nxm where $b_{ij} = a_{ji}$

Properties of transpose of a matrix:

- $(A^T)^T = A$
- $(A+B)^T = A^T + B^T$
- $(AB)^T = B^T A^T$

Singular and Nonsingular Matrix:

1. Singular Matrix: A square matrix is said to be singular matrix if its determinant is zero i.e. |A|=0
2. Nonsingular Matrix: A square matrix is said to be a non-singular matrix if its determinant is non-zero.

Properties of Matrix addition and multiplication:

1. A+B = B+A (Commutative)
2. (A+B)+C = A+ (B+C) (Associative)
3. AB ? BA (Not Commutative)
4. (AB) C = A (BC) (Associative)
5. A (B+C) = AB+AC (Distributive)

Square Matrix: A square Matrix has as many rows as it has columns. i.e. no of rows = no of columns.

Symmetric matrix: A square matrix is said to be symmetric if the transpose of the original matrix is equal to its original matrix. i.e. (A^T) = A.

Skew-symmetric: A skew-symmetric (or antisymmetric or antimetric[1]) matrix is a square matrix whose transpose equals its negative.i.e. (A^T) = -A.

Diagonal Matrix: A diagonal matrix is a matrix in which the entries outside the main diagonal are all zero. The term usually refers to square matrices.

Identity Matrix:A square matrix in which all the elements of the principal diagonal are ones and all other elements are zeros. Identity matrix is denoted as I.

Orthogonal Matrix: A matrix is said to be orthogonal if $AA^T = A^TA = I$

Idempotent Matrix: A matrix is said to be idempotent if $A^2 = A$

Involutory Matrix: A matrix is said to be Involuntary if $A^2 = I$

Note: Every Square Matrix can uniquely be expressed as the sum of a symmetric matrix and skew-symmetric matrix. $A = 1/2\,(A^T + A) + 1/2\,(A - A^T)$.

Adjoint of a square matrix: The adjoint of a matrix A is the transpose of the cofactor matrix of A

$$\text{If A} = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \text{then,}$$

$$\text{Adj A} = \text{Transpose of} \begin{bmatrix} A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \end{bmatrix} = \begin{bmatrix} A_1 & A_2 & A_3 \\ B_1 & B_2 & B_3 \\ C_1 & C_2 & C_3 \end{bmatrix}$$

$$Where, \begin{bmatrix} A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \end{bmatrix} \text{is cofactor matrix of A}$$

Properties of Adjoint:

1. $A(\text{Adj } A) = (\text{Adj } A)\, A = |A|\, I_n$
2. $\text{Adj}(AB) = (\text{Adj } B).(\text{Adj } A)$
3. $|\text{Adj } A| = |A|^{n-1}$
4. $\text{Adj}(kA) = k^{n-1}\, \text{Adj}(A)$
5. |adj(adj(A))|= |A|^(n-1)^2
6. adj(adj(A))=|A|^(n-2) * A
7. If A = [L,M,N] then adj(A) = [MN, LN, LM]
8. adj(I) = I

Where, n = number of rows = number of columns

Inverse of a square matrix:

$$A^{-1} = \frac{Adj A}{|A|}$$

Here |A| should not be equal to zero, means matrix A should be non-singular.

Properties of inverse:

1. $(A^{-1})^{-1} = A$

2. $(AB)^{-1} = B^{-1}A^{-1}$

3. only a non singular square matrix can have an inverse.

Where should we use the inverse matrix?

If you have a set of simultaneous equations:

$7x + 2y + z = 21$

$3y – z = 5$

$-3x + 4y – 2x = -1$

As we know when AX = B, then X = A-1B so we calculate the inverse of A and by multiplying it B, we can get the values of x, y, and z.

Trace of a matrix: trace of a matrix is denoted as tr(A) which is used only for square matrices and equals the sum of the diagonal elements of the matrix. Remember the trace of a matrix is also equal to the sum of eigenvalue of the matrix. For example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \mathrm{tr(A)} = 1{+}5{+}9 = 15$$

Basic Operations on Matrices:

Matrices Addition –

The addition of two matrices A_{m*n} and B_{m*n} gives a matrix C_{m*n}. The elements of C are the sum of corresponding elements in A and B.

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 12 & 14 \end{bmatrix}$$

The algorithm for addition of matrices can be written as:

```
for i in 1 to m
   for j in 1 to n
      $c_{ij} = a_{ij} + b_{ij}$
```

Time Complexity: O(n * m)

Auxiliary Space: O(n * m)

Key points:

- Addition of matrices is commutative which means A+B = B+A
- Addition of matrices is associative which means A+(B+C) = (A+B)+C
- The order of matrices A, B and A+B is always same
- If order of A and B is different, A+B can't be computed
- The complexity of addition operation is O(m*n) where m*n is order of matrices

Matrices Subtraction –

The subtraction of two matrices A_{m*n} and B_{m*n} gives a matrix C_{m*n}. The elements of C are difference of corresponding elements in A and B which can be represented as:

$$\begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$$

The algorithm for subtraction of matrices can be written as:

```
for i in 1 to m
   for j in 1 to n
      $c_{ij} = a_{ij} - b_{ij}$
```

Time Complexity: O(n * m)

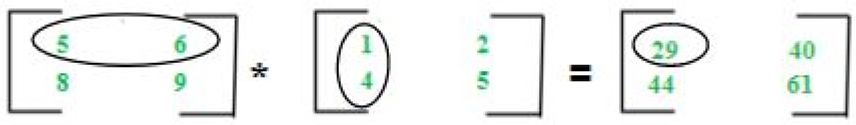
Auxiliary Space: O(n * m)

Key points:

- Subtraction of matrices is non-commutative which means A-B ≠ B-A
- Subtraction of matrices is non-associative which means A-(B-C) ≠ (A-B)-C
- The order of matrices A, B and A-B is always same
- If order of A and B is different, A-B can’t be computed
- The complexity of subtraction operation is O(m*n) where m*n is order of matrices

Matrices Multiplication –

The multiplication of two matrices A_{m*n} and B_{n*p} gives a matrix C_{m*p}. It means number of columns in A must be equal to number of rows in B to calculate C=A*B. To calculate element c_{11}, multiply elements of 1st row of A with 1st column of B and add them (5*1+6*4) which can be shown as:



The algorithm for multiplication of matrices A with order m*n and B with order n*p can be written as:

```
for i in 1 to m
   for j in 1 to p
      $c_{ij}$ = 0
      for k in 1 to n
         $c_{ij}$ += $a_{ik}$*$b_{kj}$
```

Time Complexity: O(n * m)

Auxiliary Space: O(n * m)

Key points:

- Multiplication of matrices is non-commutative which means A*B ≠ B*A
- Multiplication of matrices is associative which means A*(B*C) = (A*B)*C
- For computing A*B, the number of columns in A must be equal to number of rows in B
- Existence of A*B does not imply existence of B*A
- The complexity of multiplication operation (A*B) is O(m*n*p) where m*n and n*p are order of A and B respectively

- The order of matrix C computed as A*B is m*p where m*n and n*p are order of A and B respectively

<u>**More Resources:**</u>

Determinant, Adjoint and Inverse of Matrix:

https://www.geeksforgeeks.org/determinant-of-a-matrix/

Implementation of Basic Matrix Operations:

https://www.geeksforgeeks.org/different-operation-matrices/

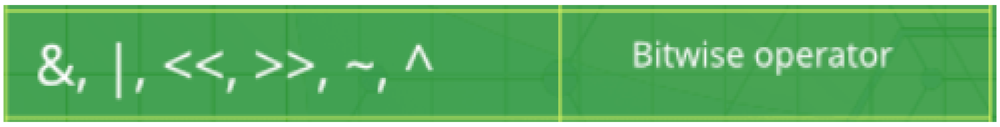
BIT MANIPULATION

Introduction:

Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a word. Computer programming tasks that require bit manipulation include low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization.

Bitwise Operators in C/C++

In C, the following 6 operators are bitwise operators (work at bit-level).

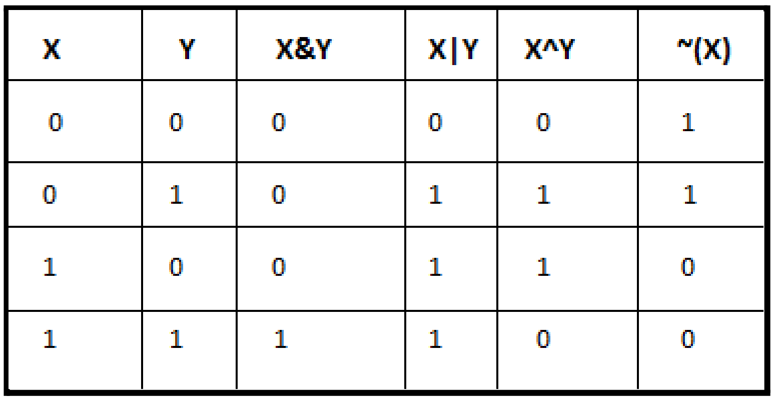


&, |, <<, >>, ~, ^	Bitwise operator

1. The **& (bitwise AND)** in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
2. The **| (bitwise OR)** in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
3. The **^ (bitwise XOR)** in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
4. The **<< (left shift)** in C or C++ takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.

5. The **>> (right shift)** in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
6. The **~ (bitwise NOT)** in C or C++ takes one number and inverts all bits of it

Truth Table For Basic Bitwise Operations:



X	Y	X&Y	X|Y	X^Y	~(X)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Table 1

Example:

```
// C Program to demonstrate use of bitwise operators
#include <stdio.h>
int main()
{
    // a = 5(00000101), b = 9(00001001)
    unsigned char a = 5, b = 9;

    // The result is 00000001
    printf("a = %d, b = %d\n", a, b);
    printf("a&b = %d\n", a & b);

    // The result is 00001101
    printf("a|b = %d\n", a | b);

    // The result is 00001100
    printf("a^b = %d\n", a ^ b);

    // The result is 11111010
    printf("~a = %d\n", a = ~a);

    // The result is 00010010
    printf("b<<1 = %d\n", b << 1);

    // The result is 00000100
    printf("b>>1 = %d\n", b >> 1);

    return 0;
}
```

Output:

```
a = 5, b = 9
a&b = 1
a|b = 13
a^b = 12
~a = 250
b<<1 = 18
b>>1 = 4
```

Explanation:

Perform the bitwise operations on each bit. (Refer to the Table 1 above (Page 20))

a = 5 = $(00000101)_2$ b = 9 = $(00001001)_2$

a & b = (00000101) & (00001001) = $(00000001)_2$ = 1

a | b = (00000101) | (00001001) = $(00001101)_2$ = 13

a ^ b = (00000101) ^ (00001001) = (00001100) = 12

~a = ~(00000101) = $(11111010)_2$ = 250

b << 1 = (00001001) << 1 = $(00010010)_2$ = 18

b >> 1 = (00001001) >> 1 = (00000100) = 4

Interesting facts about bitwise operators

1. The left shift and right shift operators should not be used for negative numbers.
2. The bitwise XOR operator is the most useful operator from a technical interview perspective.
3. The bitwise operators should not be used in place of logical operators.
4. The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively.
5. The & operator can be used to quickly check if a number is odd or even.
6. The ~ operator should be used carefully.

Example of bit manipulation

1. To determine if a number is a power of two, conceptually we may repeatedly do an integer divide by two until the number won't divide by 2 evenly; if the only factor left is 1, the original number was a power of 2. Using bit and logical operators, there is a simple expression which will return true (1) or false (0):

   ANS: `bool isPowerOfTwo = (x != 0) && ((x & (x - 1)) == 0);`

Explanation:

Method uses the fact that powers of two have one and only one bit set in their binary representation:

- x   == 0...010...0
- x-1   == 0...001...1
- x & (x-1) == 0...000...0

If the number is neither zero nor a power of two, it will have '1' in more than one place

- x   == 0...1...010...0
- x-1   == 0...1...001...1
- x & (x-1) == 0...1...000...0

The following are many other interesting problems using the XOR operator.

1. Find the Missing Number
2. swap two numbers without using a temporary variable
3. A Memory Efficient Doubly Linked List
4. Find the two non-repeating elements.
5. Find the two numbers with odd occurrences in an unsorted-array.
6. Add two numbers without using arithmetic operators.
7. Swap bits in a given number/.
8. Count the number of bits to be flipped to convert a to b .
9. Find the element that appears once.
10. Detects if two integers have opposite signs.

Important Links:

1. **Bits manipulation (Important tactics)**
2. **Bitwise Hacks for Competitive Programming**
3. **Bit Tricks for Competitive Programming**