BIT MANIPULATION

01.) MINIMUM BIT FLIPS TO CONVERT NUMBER

```cpp
class Solution {
public:
    int minBitFlips(int start, int goal) {
        return __builtin_popcount(start^goal);
    }
};
```

02.) SUBSETS

```cpp
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        int n = nums.size();
        int tot_sub = 1 << n;  // Total number of subsets is 2^n
        vector<vector<int>> ans;

        for (int i = 0; i < tot_sub; i++) {
            vector<int> temp;
            for (int j = 0; j < n; j++) {
                if (i & (1 << j)) {
                    temp.push_back(nums[j]);
                }
            }
            ans.push_back(temp);
        }
        return ans;
    }
};
```

03.) SINGLE NUMBER I

```cpp
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int ans=0;
        for(int i=0;i<nums.size();i++){
            ans=ans^nums[i];
        }
        return ans;
    }
};
```

BIT MANIPULATION

4.) SINGLE NUMBER II

CODE 01

```cpp
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int bitCount[32] = {0};

        // Count the number of 1s in each bit position
        for (int num : nums) {
            for (int i = 0; i < 32; ++i) {
                if (num & (1 << i)) {
                    bitCount[i]++;
                }
            }
        }

        // Reconstruct the single number from the bit counts
        int result = 0;
        for (int i = 0; i < 32; ++i) {
            if (bitCount[i] % 3 != 0) {
                result |= (1 << i);
            }
        }

        return result;
    }
};
```

CODE 02

```cpp
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        int n = nums.size();
        for (int i = 0; i < n; i += 3) {
            if (i == n - 1 || nums[i] != nums[i + 1]) {
                return nums[i];
            }
        }
        return -1;
    }
};
```

BIT MANIPULATION

CODE 03

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int ones = 0, twos = 0, threes = 0;

        for (int num : nums) {
            int ones_and_num = ones & num;
            twos = twos | ones_and_num;
            ones = ones ^ num;

            int ones_and_twos = ones & twos;
            threes = ones_and_twos;
            ones = ones & ~threes;
            twos = twos & ~threes;
        }

        return ones;
    }
};
```

BIT MANIPULATION

5.) SINGLE NUMBER III

CODE 01

```cpp
class Solution {
public:
    vector<int> singleNumber(vector<int>& nums) {
        long long int xorr = 0;
        for(int i = 0; i < nums.size(); i++){
            xorr = xorr ^ nums[i];
        }
        int rightmost = (xorr & (xorr - 1)) ^ xorr;
        int b1 = 0;
        int b2 = 0;
        for(int i = 0; i < nums.size(); i++){
            if(nums[i] & rightmost){
                b1 = b1 ^ nums[i];
            }
            else {
                b2 = b2 ^ nums[i];
            }
        }
        return {b1, b2};
    }
};
```

CODE 02

```cpp
class Solution {
public:
    vector<int> singleNumber(vector<int>& nums) {
        long long xorr = 0; // Use long long to avoid overflow
        for(int i = 0; i < nums.size(); i++){
            xorr = xorr ^ nums[i];
        }
        // Find the rightmost set bit using bit manipulation
        long long rightmost = xorr & -xorr;
        int b1 = 0;
        int b2 = 0;
        for(int i = 0; i < nums.size(); i++){
            if(nums[i] & rightmost)
                b1 = b1 ^ nums[i];
            else
                b2 = b2 ^ nums[i];
        }
        return {b1, b2};
    }
};
```

BIT MANIPULATION

6.) FIND XOR FROM 1 TO N

```
int computeXOR(int n) {
    // Initialize result variable
    int res = 0;

    // If n is multiple of 4
    if (n % 4 == 0)
        res = n;

    // If n % 4 gives remainder 1
    else if (n % 4 == 1)
        res = 1;

    // If n % 4 gives remainder 2
    else if (n % 4 == 2)
        res = n + 1;

    // If n % 4 gives remainder 3
    else if (n % 4 == 3)
        res = 0;

    return res;
}
```

BIT MANIPULATION

7.) FIND XOR IN GIVEN RANGE FROM L TO R

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

// Function to calculate XOR of numbers from 1 to n
int computeXOR(int n) {
    // Initialize result variable
    int res = 0;

    // If n is multiple of 4
    if (n % 4 == 0)
        res = n;

    // If n % 4 gives remainder 1
    else if (n % 4 == 1)
        res = 1;

    // If n % 4 gives remainder 2
    else if (n % 4 == 2)
        res = n + 1;

    // If n % 4 gives remainder 3
    else if (n % 4 == 3)
        res = 0;

    return res;
}

// Function to compute XOR from L to R
int rangeXOR(int L, int R) {
    return computeXOR(R) ^ computeXOR(L - 1);
}
```

BIT MANIPULATION

7.) DIVIDE TWO INTEGERS

```cpp
class Solution {
public:
    int divide(int dividend, int divisor) {
        if (dividend == INT_MIN && divisor == -1)
            return INT_MAX;

        unsigned long long dvd = abs((long long)dividend);
        unsigned long long dvs = abs((long long)divisor);

        long long sign = (dividend > 0) ^ (divisor > 0) ? -1 : 1;
        int quotient = 0;
        long long multiple = 1;

        while ((dvs << 1) <= dvd) {
            dvs <<= 1;
            multiple <<= 1;
        }

        while (dvd >= abs((long long)divisor)) {
            while (dvd >= dvs) {
                dvd -= dvs;
                quotient += multiple;
            }
            dvs >>= 1;
            multiple >>= 1;
        }

        return sign * quotient;
    }
};
```

THANK YOU !