Bit Manipulation

- Given an integer return its binary representation in a string

```
#include <bitset>

string intToBinary(int num) {
    bitset<32> binary(num);
    // specify the number of bits in <>
    return binary.to_string().substr(binary.to_string().find('1'));
    // just cut of extra starting zeroes
}

int main() {
    int num = 42;
    string binaryStr = intToBinary(num);

        int decimal = stoi(binaryStr, nullptr, 2);
        // 2 here defines the base

    return 0;
}
```

- int → 4bytes || long long → 8bytes
- number >> k ⇒ number is divided by 2^k
- number << k ⇒ number is multiplied by 2^k || 1 << n = 2^n
- INT_MAX = 2^31 - 1 || INT_MIN = -2^31
- ^ → XOR || ~ → NOT

Given two integers swap them without any extra memory.

- XOR of same numbers is 0
- XOR of 0 and number is number itself

```
int a, b;
a = (a ^ b);
b = a ^ b; => b = (a ^ b) ^ b = a
a = a ^ b; => a = (a ^ b) ^ (a) = b
```

Given an integer, check if its ith bit in binary form is set or not (set ⇒ 1)

- number & (1 << i) ≠ 0 ⇒ set
- (number >> i) & 1 == 1 ⇒ set

Given an integer set its ith bit (if already 1 do nothing)

- ((number >> i) | 1) << i
- number | (1 << i)

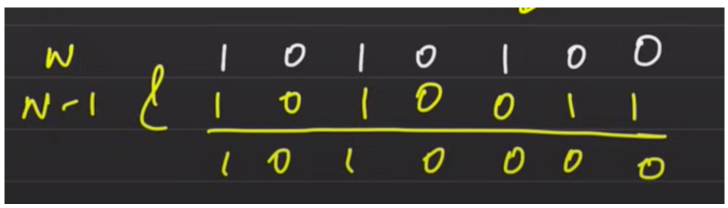
Given an integer unset its ith bit (if already 0 do nothing)

- number & ~(1 << i)
- ((number >> i) & ~1) << i

Given an integer toggle its ith bit

- number ^ (1 << i)

Given an integer unset the rightmost set bit



- number & number - 1

Find the rightmost set bit ⇒ number & ~(number - 1)

tell if a given number is power of 2

```
return (n & n-1 == 0)
```

Given a number return the number of 1 bits in it

```
int count_sets(int n) {
  int ans = 0;
  while(n > 0) {
      n = n & (n - 1);
      // we keep decreasing the right most bit
      ans++;
  }
  return ans;
}
```

```
__builtin_popcount(num)
// std function
```

- or we can keep track of how many times the & with 1 is 1 which shifting the number right

Given two integers return the minimum number of bitflips required to convert one to other

- Do the XOR of both
- Calculate the number of set bits in it {as this 1 are where the two of them differ}

Generate all possible subsets of an array {BIT MASKING}

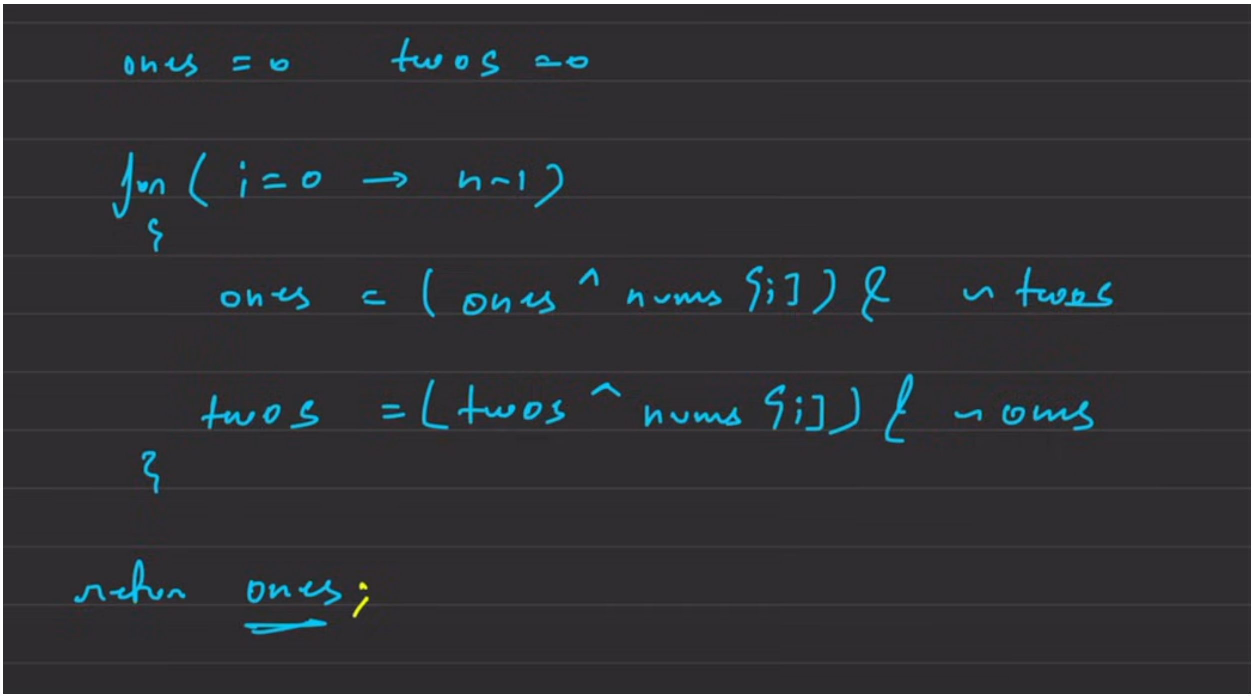
```
vector<vector<int>> subsets(vector<int>& nums) {
    int n = nums.size();
    int subsets_count = 1 << n;
    // This is 2^n, the total number of subsets
    vector<vector<int>> ans;

    for (int num = 0; num < subsets_count; num++) {
        vector<int> subset;
        for (int i = 0; i < n; i++) {
            if (num & (1 << i)) {
            // Check if the i-th bit is set
                subset.push_back(nums[i]);
            }
        }
        ans.push_back(subset);
        // Add the subset to the answer list
    }

    return ans;
}
```

Given an array all elements are repeated 3 times except one return it

```
int findUniqueNumber(const vector<int>& nums) {
    int ans = 0;

    // Iterate through each bit position
    // (0 to 31 for a 32-bit integer)
    for (int bitIndex = 0; bitIndex < 32; ++int bitIndex) {
        int cnt = 0;

        // Count how many numbers have the current bit set to 1
        for (int i = 0; i < nums.size(); ++i) {
            if (nums[i] & (1 << bitIndex)) {
                cnt++;
            }
        }

        // If count is not divisible by 3, it means the unique number
        // has a 1 bit at this position
        if (cnt % 3 == 1) {
            ans |= (1 << bitIndex);
```

```
            // it is 0 no need to specifi as we have initialised with 0
        }
    }

    return ans;
}
```

```
int findSpecialElement(std::vector<int>& nums) {
    // Step 1: Sort the array
    sort(nums.begin(), nums.end());

    int n = nums.size();

    // Step 2: Iterate through every third element
    for (int i = 1; i < n; i += 3) {
        // Step 3: Check if the current element equals the previous one
        if (nums[i] == nums[i - 1]) {
            return nums[i - 1];  // Return the element if found
        }
    }

    // Step 4: If no match found, return the last element
    return nums[n - 1];
}
```



- we need not declare a three’s bucket

Given a nums arrays where all numbers appear in pair, but there are two numbers which are single

- return those two

```
vector<int> singleNumber(vector<int>& nums) {
    long xorr = 0; // why long ? explained later

    for (int i = 0; i < nums.size(); i++) {
        xorr = xorr ^ nums[i];
    }

    // Declare a variable to store
    // the rightmost bit set in xorr
    int = rightmost = (xorr & (xorr - 1)) ^ xorr;
    // if in case the extra elements are 0 and INT_MIN,
    // then xorr-1 can't stay int int range hence use a long

    // Initialize buckets to store numbers that
    // are set at the corresponding bit or not
    int bucket1 = 0;
    int bucket2 = 0;

    // Loop through all elements in nums
    for (int i = 0; i < nums.size(); i++) {
        // Check if the rightmost bit set
        // in nums[i] is also set in rightmost
        if (nums[i] & rightmost) {
            // XOR nums[i] with bucket1
            bucket1 = nums[i] ^ bucket1;

        // If the rightmost bit set in
        // nums[i] is not set in rightmost
        } else {
             // XOR nums[i] with bucket2
            bucket2 = nums[i] ^ bucket2;
        }
    }
    // Return the two unique numbers
    // found in bucket1 and bucket2
    return {bucket1, bucket2};
}
```

XOR of numbers in range L - R

```
int xorTill(int n){ // Pattern to be observed
    if(n%4 == 1){
        return 1;
    }
```

```
    else if(n%4 == 2){
        return n+1;
    }
    else if(n%4 == 3){
        return 0;
    }
    else{
        return n;
    }
}

int xorInRange(int L, int R){
    // Compute XOR of numbers from 1 to L-1
    // and 1 to R using the xorTill function
    int xorTillL = xorTill(L-1);
    int xorTillR = xorTill(R);
    // Compute XOR of the range from L to R
    return xorTillL ^ xorTillR;
}
```

Divide Two Integers without using Multiplication and Division Operators

```
int divide(int dividend, int divisor) {
    // Check if dividend and divisor
    // are equal, return 1 if true
    if(dividend == divisor){
        return 1;
    }

    // Determine the sign of the result,
    // true for positive, false for negative
    bool sign = true;

    if(dividend >= 0 && divisor < 0){
        sign = false;
    }
    else if(dividend <= 0 && divisor > 0){
        sign = false;
    }

    // Take absolute values
    // of dividend and divisor
    long n = abs(dividend);
    long d = abs(divisor);

    // Store original divisor absolute
    // value in divisor variable
```

```
    divisor = abs(divisor);

    // Initialize quotient to 0
    long quotient = 0;

    // Perform division using
    // repeated subtraction
    while(n >= d){
        // Count how many times divisor can
        // be doubled before exceeding dividend
        int cnt = 0;
        while(n >= (d << (cnt+1))){
            cnt += 1;
        }
        // Add the value corresponding
        // to the current doubling to the quotient
        quotient += 1 << cnt;
        // Subtract the product of divisor
        // and the doubled value from dividend
        n -= (d << cnt);
    }

    // Handle overflow cases
    // If quotient equals (2^31) and the result
    // is supposed to be positive, return INT_MAX
    if(quotient == (1<<31)&&sign){
        return INT_MAX;
    }
    // If quotient equals (2^31) and the result
    // is supposed to be negative, return INT_MIN
    if(quotient == (1 << 31)&& !sign){
        return INT_MIN;
    }
    // Return the quotient with correct sign
    return sign ? quotient: -quotient;
}
```