Part 1 basic bitwise operators

AND &:

- Returns 1 for a bit position if both corresponding bits of the operands are 1.

OR (|):

- Returns 1 for a bit position if at least one of the corresponding bits of the operands is 1.

XOR (^):

- Returns 1 for a bit position if the corresponding bits of the operands are different.

NOT (~):

- Flips all the bits of the operand.

Left Shift (<<):

- Shifts bits to the left, filling with 0s on the right.

Right Shift (>>):

- Shifts bits to the right. Behavior on sign bit varies (logical vs arithmetic shift).

Part 2 Usages

- ### Swap Function

- Variable swapping using bitwise operations allows you to exchange the values of two variables without using additional variables

- Increasing efficiency and saving memory space

Step1: a = a ^ b: a becomes a ^ b
Step2: b = a ^ b: We get (a^b) ^ b = a ^(b^b) = a^0 = a, effectively swapping "a" into "b".
Step3: a = a ^ b: We have a^b = (a^b)^a = a^b^a = a^a^b = 0^b = b, which effectively swaps "b" into "a".

```
void swap(int *a, int *b) {
    *a = *a ^ *b;
    *b = *a ^ *b;
    *a = *a ^ *b;
}
```

- ### Hamming Distance

- The Hamming distance is a measure used to quantify the differences between two strings of equal length.

- It is commonly employed in the processing and detection of video images

Step1: Calculate the XOR of x and y.
Step2: Initialize a variable to store the distance.

Step3: Start a loop until xor becomes 0.
Step4: Check the rightmost bit of xor and add it to the distance.
Step5: Right-shift xor by 1 to discard the rightmost bit.
Step6: Return the calculated Hamming distance.

```
int hammingDistance(int x, int y) {
    int xor = x ^ y;
    int distance = 0;
    while (xor) {
        distance += xor & 1;
        xor >>= 1;
    }
    return distance;
}
```

- Exponentiation by squaring

- The Fast Power Algorithm efficiently calculates powers

- Especially for large integers, it leverages binary exponentiation and bitwise operations.

1. Converts an exponent to a binary representation.
2. Starting with the highest bit, iterate through the binary representation of the exponent.
3. For each bit:

If it is 1, multiply the base (base) by the current digit power of 2 and accumulate to the result.
If it is 0, continue traversing the next bit.

4. The final result obtained is the desired power.

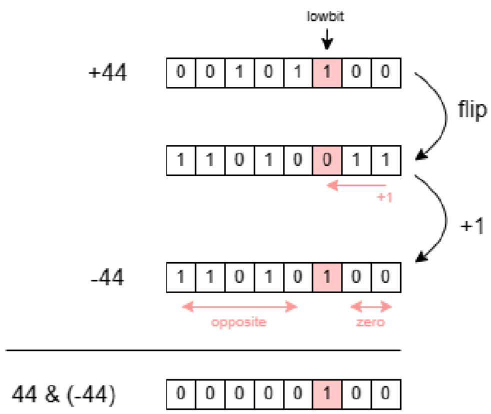
```
long long fast_power(int base, int exponent) {
    long long result = 1;
    long long power = base;

    while (exponent > 0) {
        if (exponent & 1) {
            result *= power;
        }
        power *= power;
        exponent >>= 1; // right shift == divide by 2
    }

    return result;
}
```

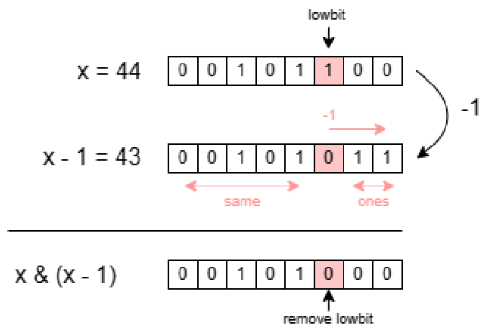
Part 3 Lowbit(x)

- lowbit(x) = the rightmost bit “1” for an integer x
- lowbit(x) stands for the smallest element in the set x.

lowbit(x) = x & (-x)

Remove lowbit: x & (x - 1)





Part 4 Single Number

260. Single Number III

Solved

Medium | Topics | Companies

Given an integer array `nums`, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once. You can return the answer in **any order**.

You must write an algorithm that runs in linear runtime complexity and uses only constant extra space.

- Take advantage of the key features of XOR and lowbit
- Strategy:
  - K = X^Y, K!=0
  - lowbit K'=K&(~K+1)
  - two groups:
    elements in group 1&K'=0;
    elements in group 2&K'=1
  - XOR all elements in group 1 to get X
  - Y=K^X
- Code Implementation:

```c
int* singleNumber(int* nums, int numsSize, int* returnSize) {
    int K = 0, lowbit;
    int* res = ( int* ) malloc( 2 * sizeof( int ));
    res[0]=0;
    res[1]=0;
    for ( int i = 0; i < numsSize; i++ )
        K=K^nums[i];
    lowbit = K & ( ~K + 1 );
    for ( int i = 0; i < numsSize; i++ ){
        if( ( lowbit & nums[i] ) != 0 )
            res[0] = res[0] ^ nums[i];
    }
    res[1] = K ^ res[0];
    * returnSize = 2;
    return res;
```

Part 5 Combination

> ### 77. Combinations
>
> Medium | Topics | Companies
>
> Given two integers `n` and `k`, return *all possible combinations of* `k` *numbers chosen from the range* `[1, n]`.
>
> You may return the answer in **any order**.
>
> **Example 1:**
>
> **Input:** n = 4, k = 2
> **Output:** [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]

we can use 1 to represent the number we chose and 0 that we did not choose.

Algorithm
1.Iterate all possible subset((1 << n) – 1)
2.If there are k 1's in the bits
3. Shift i bits to the right. If the position is 1, it means that the number (i + 1) is selected into the combination.(if (length >> i) & 1)