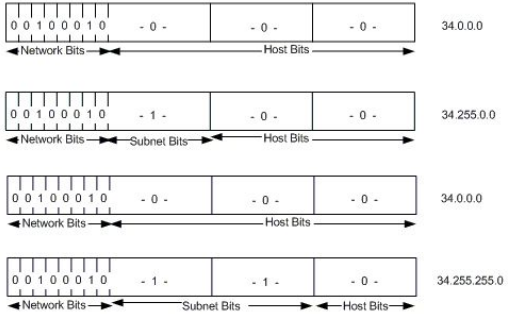
Bit Manipulation

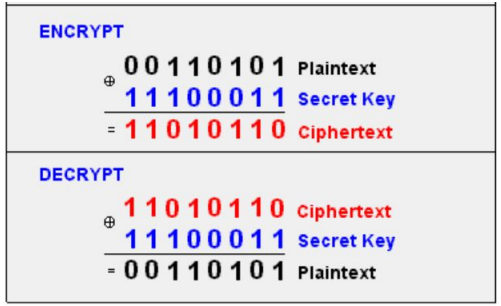
Northeastern University
Khoury College of
Computer Sciences

Project Members:
Ruyi Bao, Ziyu Huang, Xinrui Yi,
Kaichen Qu, Yichi Zhang

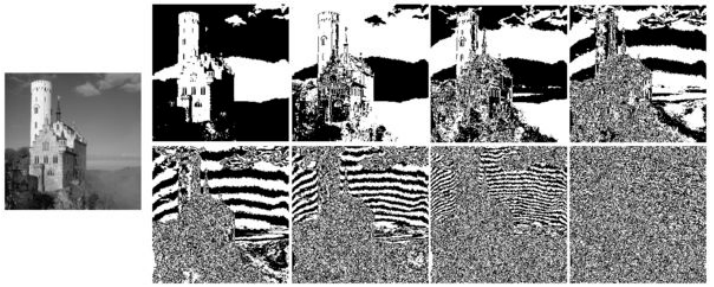
Introduction



Network Protocal



Cryptography



Digital Image Processing

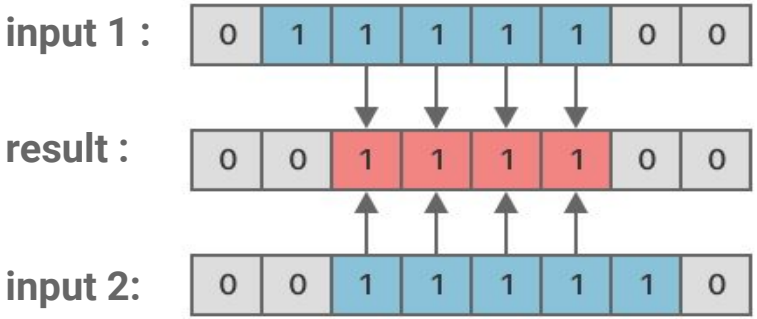
Northeastern University
Khoury College of
Computer Sciences

Outline

1. Bit operator
2. Bit application
3. Leetcode 77
4. Leetcode 231
5. Leetcode 260

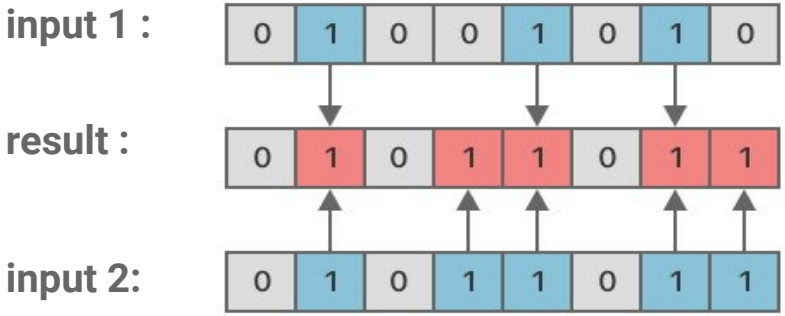
Bitwise – AND &

1 & 1 = 1
1 & 0 = 0
0 & 0 = 0



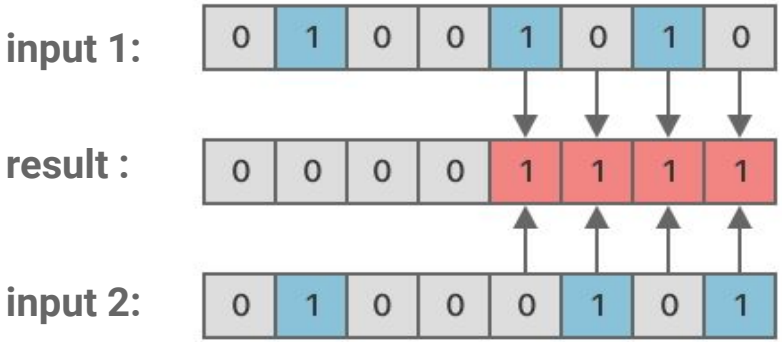
Bitwise – OR |

1 | 1 = 1
1 | 0 = 1
0 | 0 = 0



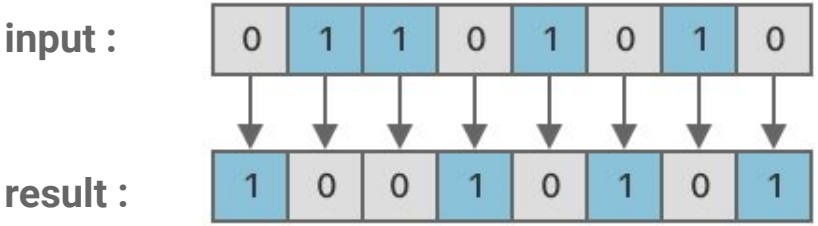
Bitwise – XOR ^

1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 0 = 0

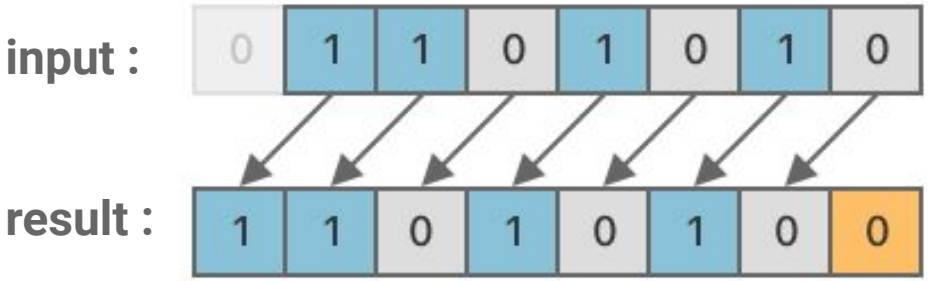


Bitwise – NOT ~

~1 = 0
~0 = 1

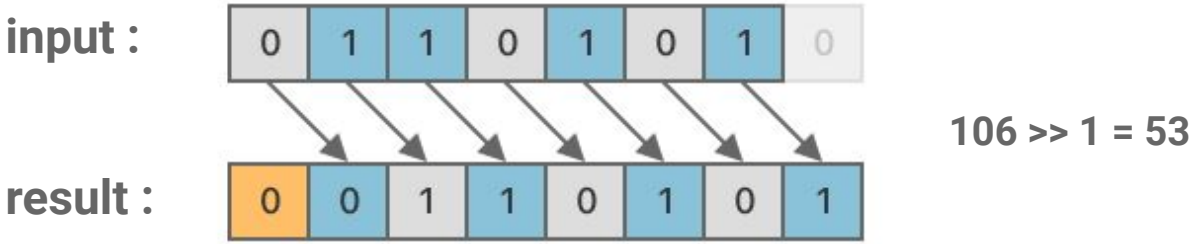


Bit Operators – Left Shift <<



106 << 1 = 212

Bit Operators – Right Shift >>



106 >> 1 = 53

Bit operation usage

- Swap Function

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

- It increase efficiency and saving memory spaces in computer

- 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

- Exponentiation by squaring

- The Fast Power Algorithm efficiently calculates powers of 2

- Especially for large integers, it quickly and efficiently utilizes binary exponentiation and bitwise operations.

Northeastern University
Khoury College of Computer Sciences

Bit operation usage 1

- **Swap Function**

1. a = a ^ b: a becomes a ^ b
2. b = a ^ b: We get (a^b) ^ b = a ^(b^b) = a^0 = a, effectively swapping "a" into "b".
3. a = a ^ b: We have a^b = (a^b)^a = a^b^a = a^a^b = 0^b = b, which effectively swapping "b" into "a".

Swap function

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

Fast & No temporary variable

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

Bit operation usage 2

- **Hamming Distance**

1. Calculate the XOR of x and y.
2. Initialize a variable to store the distance.
3. Start a loop until xor becomes 0.
4. Check the rightmost bit of xor and add it to the distance.
5. Right-shift xor by 1 to discard the rightmost bit.
6. 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;
}
```

X ^ Y

X >> 1

Bit operation usage 3

- **Exponentiation by squaring**

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:

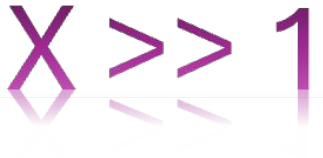
- If it is 1, multiply the base (base) by the current digit power of 2 and accumulate to the result.
- If it is 0, continue traversing the next bit.

4. Finally return the result

```
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;
}
```



Lowbit operation

What is lowbit?

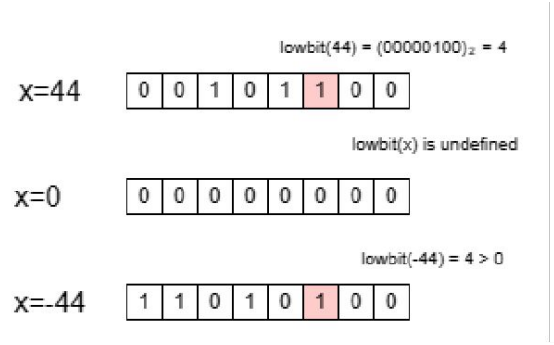
lowbit(x) = the rightmost bit "1" for an integer x

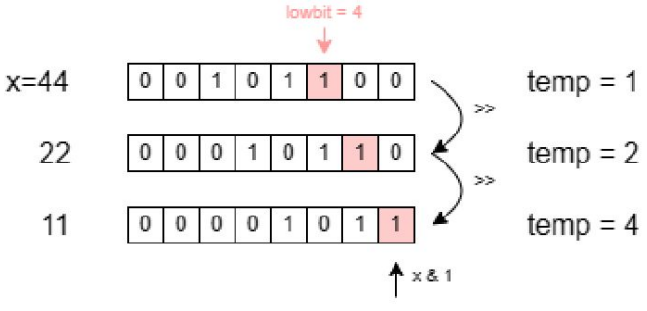
Why do we care about the lowbit?

lowbit(x) stands for the smallest element in the set x.
Function domain: positive integer

How to calculate lowbit(x)?

O(logN): while-loop + right shift operation
O(1): lowbit(x) = x & (-x)





Lowbit(x) = x & (-x)

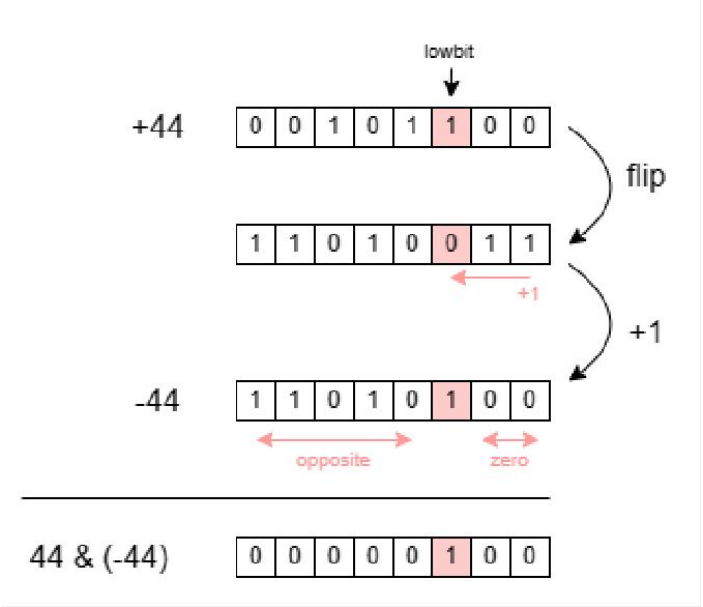
Why can we calculate lowbit like that?

2’s complement representation

After fliping, lowbit(x) becomes the rightmost "0".

When adding 1, only the bits to the right of the lowbit, including the lowbit itself, are affected. The bit carry operation halts precisely at the lowbit, as it is the rightmost '0' at that point.

Bits to the left of the lowbit are flipped, while those on the right remain unchanged, all being '0's.

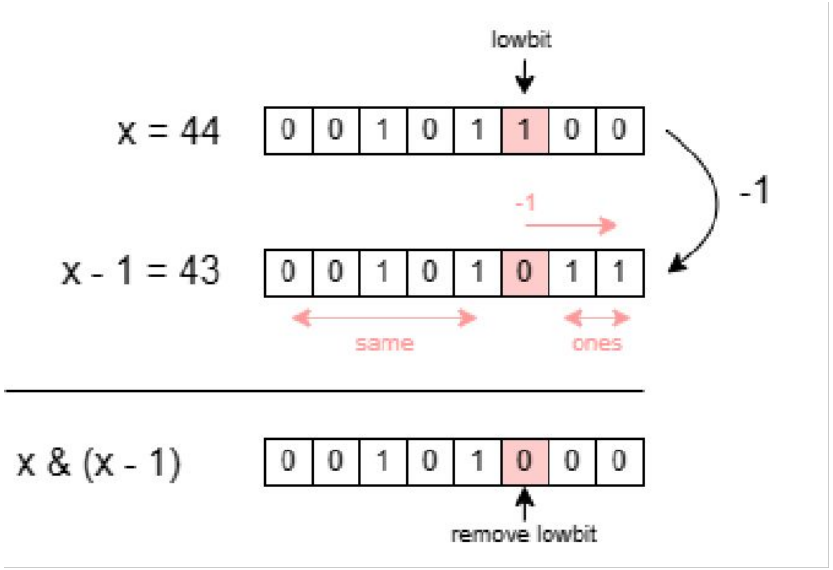


Remove lowbit: x & (x - 1)

How can we remove lowbit?

Because lowbit(x) is the rightmost bit set to '1', subtracting 1 from x affects all bits to the right of the lowbit, including the lowbit itself. The bits to the left of lowbit(x) remain unchanged.

By performing the '&' operation between x and (x - 1), lowbit(x) and all bits to its right are reset to zero, while the remaining bits are left unchanged.



LC 231 - Power of Two

Given an integer n, return if it is a power of two.

While-loop Solution - O(log n)

True case: 16 = 00001000 - only 1 bit "1"
False case: 19 = 00001011

O(1) Solution:
n == lowbit(n)
n & (n-1) == 0

```
bool isPowerOfTwo(int n) {
    if (n <= 0) return false;
    while (n % 2 == 0) {
        n >>= 1;
    }
    return n == 1;
}
```

```
bool isPowerOfTwo(int n) {
    return n <= 0 ? false : n == (n & -n);
}
```

```
bool isPowerOfTwo(int n) {
    return n <= 0 ? false : (n & (n - 1)) == 0;
}
```

LC 260 - Single Number III

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.

Space Complexity : O(1), excluding Hashmap

LC 136 - Single Number

Given a non-empty array of integers nums, every element appears *twice* except for one. Find that single one.

```
int singleNumber(int* nums, int numsSize) {
    int res = 0;
    for (int i = 0; i < numsSize; i++) {
        res ^= nums[i];
    }
    return res;
}
```

Strategy & Implimentation

Assuming the two elements that appear only once are X and Y, we aim to divide these numbers into two groups such that X and Y appear in separate groups, and all other numbers appear in pairs within each group.

1. K = X^Y, K!=0
2. lowbit K'=K&(~K+1)
3. two groups:

   ele in group 1&K'=0;

   ele in group 2&K'=1
4. XOR all ele in group 1 to get X
5. Y=K^X

```
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;
```

LC 77- Combinations

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]]
```

combination	bit
1,2	0011
1,3	0101
1,4	1001
2,3	0110

Algorithm
1.Iterate all possible subset((1 << n) – 1)
2.If there are k 1's in the bits(n&(n-1))
3.iterates over the bits of length and checks if each bit is 1. If the position is 1, it means that the number (i + 1) is selected into the combination.(if (length >> i) & 1)

LC 77- Combinations

```c
int countSetBits(int n) {
    int count = 0;
    while (n) {
        n = n & (n - 1);
        count++;
    }
    return count;
}
int** combine(int n, int k, int* returnSize, int** returnColumnSizes) {
    int totalSubsets = (1 << n)-1;    // Calculate the total number of subsets (2^n)
    int** res = (int**)malloc(sizeof(int*) * totalSubsets);    // Allocate memory for storing combinations
    *returnColumnSizes = (int*)malloc(sizeof(int) * totalSubsets);
    *returnSize = 0;
    for (int length = 1; length <= totalSubsets; length++) {    // Iterate over all possible subsets
        if (countSetBits(length) == k) {        // Check if the number of set bits in 'length' is equal to k
            res[*returnSize] = (int*)malloc(sizeof(int) * k);           // Allocate memory for the current combination
            int index = 0;
            for (int i = 0; i < n; i++) {           // Extract elements based on set bits in 'length'
                if (length & (1 << i)) {
                    res[*returnSize][index++] = i + 1;
                }
            }
            // Set the size of the current combination
            (*returnColumnSizes)[*returnSize] = k;
            (*returnSize)++;
        }
    }
    return res;
}
```

n=4 k=2

total subsets=1111=15

from 0001 to 1111

length=3 0011
[1,2]

[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]

Time complexity:
O(k*2^n)

Space complexity:
O(k*2^n)

Northeastern University
Khoury College of
Computer Sciences

Conclusion

- Bit operator
  - AND(&), OR(|), XOR(^) ,NOT(~),LEFT/RIGHT SHIFT(<<,>>)
- Bit application
  - Swap
  - Hamming Distance
  - Exponentiation by squaring
  - Low bit
  - Masking
- Advantage
  - Efficiency
  - Memory Optimization

Thank you for listening!

Any Questions?