Bitwise operations

Numbers representation

- Numbers in computer represented in binary numeral system.
- Every decimal number you see at output or give to program by yourself converted to binary.
- There are number of various binary representations for signed numbers.

Numbers representation

- 3 basic types of numbers representation in computer:
  - Sign and magnitude method.
  - Ones' complement.
  - Two's complement.

Numbers representation
Sign and magnitude method

- Simple representation with power of 2s for both signed and unsigned numbers. First bit are used for sign, 0 for positive numbers, 1 for negative:
  - $13_{10} = 0001\ 1101_2$
  - $-13_{10} = 1001\ 1101_2$
  - $91_{10} = 0101\ 1011_2$
  - $-91_{10} = 1101\ 1011_2$

Numbers representation
Sign and magnitude method

- Advantages:
  - Easy to calculate signed and unsigned numbers representation.
- Disadvantages:
  - There are two zeroes in this representation:
    - $0_{10} = 0000\ 0000_2$
    - $-0_{10} = 1000\ 0000_2$

  So the range of numbers this method allows to represent is $[-(2^n - 1);\ 2^n - 1]$

  - Hard to perform even standard arithmetic operations with signed and unsigned numbers.
    - $12_{10} - 12_{10} = 0000\ 1100_2 + 1000\ 1100_2 = {}^{+}\begin{array}{r} 00001100 \\ \underline{10001100} \\ 10011000 \end{array} = 1001\ 1000_2 = -24_{10} \neq 12_{10} - 12_{10}$

Numbers representation
Ones' complement

- Positive numbers are the same simple, negative values are the bit $complement$ of the corresponding positive value:
  - $13_{10} = 0001\ 1101_2$
  - $-13_{10} = 1110\ 0010_2$
  - $91_{10} = 0101\ 1011_2$
  - $-91_{10} = 1010\ 0100_2$

Numbers representation
Ones' complement

- Advantages:
  - Easy to perform standard arithmetic operations with signed and unsigned numbers:
    - $12_{10} + 23_{10} = 0000\ 1100_2 + 0001\ 0111_2 = {}^{+}\begin{array}{r} 00001100 \\ \underline{00010111} \\ 00100011 \end{array} = 0010\ 0011_2 = 35_{10}$
    - $12_{10} - 23_{10} = 0000\ 1100_2 + 1110\ 1000_2 = {}^{+}\begin{array}{r} 00001100 \\ \underline{11101000} \\ 11110100 \end{array} = 1111\ 0100_2 = -11_{10}$
- Disadvantages:
  - There are two zeroes in this representation:
    - $0_{10} = 0000\ 0000_2$
    - $-0_{10} = 1111\ 1111_2$

  So the range of numbers this method allows to represent is $[-(2^n - 1);\ 2^n - 1]$

Numbers representation
Two's complement

- Positive numbers are the same simple, negative values are the bit complement of the corresponding positive value, the value of 1 is then added to the resulting value, ignoring the overflow which occurs when taking the two's complement of 0:
  - $13_{10} = 0001\ 1101_2$
  - $-13_{10} = 1110\ 0011_2$
  - $91_{10} = 0101\ 1011_2$
  - $-91_{10} = 1010\ 0101_2$
  - $0_{10} = -0_{10} = 0000\ 0000_2$

Numbers representation
Two's complement

- Advantages:
  - Easy to perform standard arithmetic operations with signed and unsigned numbers
    - $12_{10} + 23_{10} = 0000\ 1100_2 + 0001\ 0111_2 = {}^{+}\begin{array}{c} 00001100 \\ \underline{00010111} \\ 00100011 \end{array} = 0010\ 0011_2 = 35_{10}$
    - $12_{10} - 23_{10} = 0000\ 1100_2 + 1110\ 1001_2 = {}^{+}\begin{array}{c} 00001100 \\ \underline{11101001} \\ 11110101 \end{array} = 1111\ 0101_2 = -\ 0000\ 1011_2 = -11_{10}$
  - There is only one zero in this representation:
    - $0_{10} = -0_{10} = 0000\ 0000_2$

So the range of numbers this method allows to represent is $[-2^n;\ 2^n - 1]$
Modern computers uses exactly this number representation.

Bitwise operations

- **Bitwise operators** are a binary operators and treat their operands as a sequence of N bits (zeroes and ones), rather than as decimal numbers. For example, the decimal number 9 has a binary representation of 1001. Bitwise operators perform their operations on such binary representations.

- Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on.

- The operator is applied to each pair of bits, and the result is constructed bitwise.

Bitwise operations

- The following table summarizes all bitwise operators:

Operation name	Operation operator
OR	|
AND	&
XOR	^
RIGHT SHIFT	>>
LEFT SHIFT	<<
COMPLEMENT, NOT	~

Bitwise operations
& (AND)

- Performs the AND operation on each pair of bits.
- x AND y yields 1 only if both x and y are 1. The truth table for the AND operation is:

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise operations
& (AND)

- Examples:

$$12_{10} \,\&\, 23_{10} = 0000\ 1100_2 \,\&\, 0001\ 0111_2 = \& \frac{\begin{matrix}00001100\\ 00010111\end{matrix}}{00000100} = 0000\ 0100_2 = 4_{10}$$

$$51_{10} \,\&\, 25_{10} = 0011\ 0011_2 \,\&\, 0001\ 1001_2 = \& \frac{\begin{matrix}00110011\\ 00011001\end{matrix}}{00010001} = 0001\ 0001_2 = 17_{10}$$

Bitwise operations
| (OR)

- Performs the OR operation on each pair of bits.
- x OR y yields 1 only if either x or y are 1. The truth table for the OR operation is

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise operations
| (OR)

- Examples:

$$12_{10} \mid 23_{10} = 0000\ 1100_2 \mid 0001\ 0111_2 = \begin{array}{r} 00001100 \\ \mid\ \underline{00010111} \\ 00011111 \end{array} = 0001\ 1111_2 = 31_{10}$$

$$51_{10} \mid 25_{10} = 0011\ 0011_2 \mid 0001\ 1001_2 = \begin{array}{r} 00110011 \\ \mid\ \underline{00011001} \\ 00111011 \end{array} = 0011\ 1011_2 = 59_{10}$$

Bitwise operations
^ (XOR)

- Performs the XOR operation on each pair of bits.
- x XOR y yields 1 if x and y are different. The truth table for the XOR operation is

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise operations
^ (XOR)

- Examples:

$$12_{10} \wedge 23_{10} = 0000\ 1100_2 \wedge 0001\ 0111_2 = {}^{\wedge}\,\frac{\begin{array}{c}00001100\\00010111\end{array}}{00011011} = 0001\ 1011_2 = 27_{10}$$

$$51_{10} \wedge 25_{10} = 0011\ 0011_2 \wedge 0001\ 1001_2 = {}^{\wedge}\,\frac{\begin{array}{c}00110011\\00011001\end{array}}{00101010} = 0010\ 1010_2 = 42_{10}$$

Bitwise operations
Relations

- The relations table for AND, OR and XOR is:

<table>
<tr><th>&</th><th>|</th><th>^</th></tr>
<tr><td>x&y == y&x</td><td>x|y == y|x</td><td>x^y == y^x</td></tr>
<tr><td>(x&y)&z == x&(y&z)</td><td>(x|y)|z == x|(y|z)</td><td>(x^y)^z == x^(y^z)</td></tr>
<tr><td>x & 0 == 0</td><td>x | 0 == x</td><td>x ^ 0 == x</td></tr>
<tr><td>x & x = x</td><td>x | x = x</td><td>x ^ x = 0</td></tr>
</table>

Bitwise operations
~ (COMPLEMENT, NOT)

- Performs the NOT operation on each bit.
- NOT x yields the inverted value (a.k.a. one's complement) of x. The truth table for the NOT operation is:

x	~x
0	1
1	0

- Examples:

  $\sim 23_{10} = \sim 0001\ 0111_2 = 1110\ 1000_2$

  $\sim 25_{10} = \sim 0001\ 1001_2 = 1110\ 0110_2$

Bitwise operations
<< (LEFT SHIFT)

- This operator shifts the first operand the specified number of bits to the left.
- Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.
- Example:
  - $12_{10} << 2_{10} = 0000\ 1100_2 << 2_{10} = 0011\ 0000_2 = 48_{10}$
  - $27_{10} << 3_{10} = 0001\ 1011_2 << 3_{10} = 1101\ 1000_2 = 216_{10}$
  - $37_{10} << 4_{10} = 0010\ 0101_2 << 4_{10} = 0101\ 0000_2 = 80_{10}$

Bitwise operations
>> (RIGHT SHIFT)

- This operator shifts the first operand the specified number of bits to the right.

- Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.

- Example:
  - $12_{10} >> 2_{10} = 0000\ 1100_2 >> 2_{10} = 0000\ 0011_2 = 3_{10}$
  - $27_{10} >> 3_{10} = 0001\ 1011_2 >> 3_{10} = 0000\ 0011_2 = 3_{10}$
  - $37_{10} >> 4_{10} = 0010\ 0101_2 >> 4_{10} = 0000\ 0010_2 = 2_{10}$

Bitwise operations Manipulations

- i-th power of 2:
  - 1 << i
- Change i-th bit of number n to 1:
  - n = n | (1 << i);
- Change i-th bit of number n to 0:
  - n = n & ~(1 << i);
- Toggle i-th bit of number n:
  - n = n ^ (1 << i);
- Check if i-th bit is 1:
  - if(n & (1 << i) != 0)

Bitwise operations
Problem: Single number

- You are given an array of integers. Every element appears twice, except for one. You need to find the element that appears only one time. Your solution should have a linear runtime complexity (O(n)). Try to implement it without using extra memory.
- Examples:

Input	Output
5 3 5 2 2 3	5
7 2 3 4 1 1 3 4	2

Bitwise operations
Problem: Single number

- Solution is following: You just need to XOR all the numbers with each other. Result will be the answer.
- Examples:

Input	Output
5 3 5 2 2 3	5
7 2 3 4 1 1 3 4	2

- $3 \wedge 5 \wedge 2 \wedge 2 \wedge 3 = 2 \wedge 2 \wedge 3 \wedge 3 \wedge 5 = 0 \wedge 0 \wedge 5 = 5$
- $2 \wedge 3 \wedge 4 \wedge 1 \wedge 1 \wedge 3 \wedge 4 = 1 \wedge 1 \wedge 2 \wedge 3 \wedge 3 \wedge 4 \wedge 4 = 0 \wedge 2 \wedge 0 \wedge 0 = 2$

Bitwise operations
Problem: Subsets

- Generate all subsets of the given set
- Examples:

Input	Output
3 1 2 5	{} {1} {2} {5} {1, 2} {2, 5} {1, 5} {1, 2, 5}

Bitwise operations
Problem: Subsets

- Solution:

```
void subsets(const std::vector<int>& set)
{
    int n = set.size();
    for (int mask = 0; mask < (1 << n); ++mask)
    {
        bool first = true;
        std::cout << "{";
        for (int i = 0; i < n; ++i)
        {
            if (mask & (1 << i))
            {
                if (!first)
                    std::cout << ", ";
                first = false;
                std::cout << set[i];
            }
        }
        std::cout << "}\n";
    }
}
```

Bitwise operations
Problem: K-Subsets

- Generate all subsets of length K of the given set
- Examples:

Input	Output
5 2 1 2 3 4 5	1 2 1 3 1 4 1 5 2 3 2 4 2 5 3 4 3 5 4 5

Bitwise operations Problem: K-Subsets

- Solution:

```
void subsets_of_lenght_K(const std::vector<int>& set, int k)
{
    int n = set.size();
    for (int mask = 0; mask < (1 << n); ++mask)
    {
        int num = 0;
        for (int i = 0; i < n; ++i)
            if (mask & (1 << i))
                ++num;
        if (num == k)
        {
            for (int i = 0; i < n; ++i)
                if (mask & (1 << i))
                    std::cout << set[i] << " ";
            std::cout << "\n";
        }
    }
}
```