Bitmasking

• What is Bitmasking?

In computer programming, the process of modifying and utilizing binary representations of numbers or any other data is known as bitmasking.

A binary digit is used as a flag in bitmasking to denote the status or existence of a feature or trait. To accomplish this, certain bits within a binary number are set or reset to reflect a particular state or value.

Bit masking is visualizing a number or other data in binary representation. Some bits are set and others are unset where set means true or 1 and unset means false or 0. It allows us to store multiple values inside one numerical variable. You should think of every bit as a separate value instead of thinking of this number as a whole.

Some of the most commonly used bitwise operations in bitmasking are:

OR (|) – sets a bit to 1 if either of the corresponding bits in the operands is 1.

AND (&) – sets a bit to 1 if both the corresponding bits in the operands are 1.

XOR (^) – sets a bit to 1 if the corresponding bits in the operands are different.

NOT (~) – flips the bits in the operand, i.e., sets 0 bits to 1 and 1 bits to 0.

Bitmasking is an operation in which we only allow certain bits to pass through while masking other bits.

- **To set a bit in a bitmask, you can use the bitwise OR operator** with a value that has a 1 in the position of the bit you want to set and 0s in all other positions. For example, to set the third bit in a bitmask, you can use the expression:

  bitmask |= (1 << 2);

This sets the third bit by shifting the value 1 two positions to the left, so that it has a 1 in the third position and 0s in all other positions. The bitwise OR operator then combines this value with the original bitmask, setting the third bit to 1 while leaving all other bits unchanged.

- **To clear a bit in a bitmask, you can use the bitwise AND operator** with a value that has a 0 in the position of the bit you want to clear and 1s in all other positions. For example, to clear the fourth bit in a bitmask, you can use the expression:

  bitmask &= ~(1 << 3);

This clears the fourth bit by first shifting the value 1 three positions to the left, so that it has a 1 in the fourth position and 0s in all other positions. The bitwise NOT operator then flips all the bits in this value, so that it has a 0 in the fourth position and 1s in all other positions. Finally, the bitwise AND operator combines this value with the original bitmask, clearing the fourth bit while leaving all other bits unchanged.

- **To check if a bit is set in a bitmask, you can use the bitwise AND operator** with a value that has a 1 in the position of the bit you want to check and 0s in all other positions. For example, to check if the second bit in a bitmask is set, you can use the expression:

  bool is_set = (bitmask & (1 << 1)) != 0;

This checks the second bit by shifting the value 1 one position to the left, so that it has a 1 in the second position and 0s in all other positions. The bitwise AND operator then combines this value with the original bitmask, resulting in a value that has 1s in all positions except the second position if the second bit is set, or 0s in all positions if it is not set. The expression then compares this value to 0 to determine if the second bit is set.

For example: Let us say we have a number 10, the binary representation of the number (10)10 is (1010). Now let us say we want to find whether the third bit is set or not. To solve this problem we will make use of the bitwise AND operator. So if we need to find whether the third bit is set or not, we are not interested in the other bits. So if we move from the least significant digit to the most (right to left) we are not interested in the first(0), second(1), and fourth(1) bit – so we will be masking these. Masking means that we will apply the bitwise AND operation with 0.

If you have a bit 'x' (0/1) and you AND it with 0, you will always get 0. If you have a bit 'x' (0/1) and you AND it with 1, you will always get 1 – For example: if x = 0 then 0 & 1 = 0 (which is x), if x = 1 the 1 & 1 = 1 (which is x).

The bits we want to pass through, we will use bitwise AND with 1. So, for the above example, we will bitwise AND 1010 with 0100. Note: Only the third bit from left is 1, the rest are 0s. When you do the bitwise AND if you get a number other than 0 then that particular bit is set, otherwise not set. For our example, we will get 0 (1010 & 0100 = 0000) that means that the third bit was not set.

If we want to check if the kth bit is set or not then we will left shift 1, k-1 times (1 << (k-1)). So, if we want to find the mask of the 1st bit we will left shift 1 by 1 – 1 = 0. So, it's 0001. For our example ((10)10), (1010) & (0001) = (0000), so we know that the first bit is not set. Let us check the second(1) bit. To find the mask we will left shift 1 by 1(2 – 1) bit, so we will get 0010. Now if we do 1010 & 0010, we get (0010) which is non-zero so we know that the second bit is set.

• Applications of Bitmasking:

Bitmasking is an effective method with numerous uses in computer science and technologies, such as:

1. **Optimization:** Algorithm optimization can be achieved using bitmasking, which substitutes bit-level operations for expensive ones. As an illustration, right shifting by one is a substantially faster operation than dividing by two.

2. **Memory efficiency:** There are several circumstances in which storing data as a bitmask can be more memory-efficient than utilizing conventional data structures. For instance, you can use a single bitmask to indicate the presence or absence of a collection of items or list rather than an array of Boolean values.

3. **Data compression:** Bitmasking can be utilized in data compression methods to minimize the size of data by encoding data as a series of bits.

4. **Graphics:** In computer graphics, bitmasking is used to alter pixels and run various operations on images.

5. **Networking:** Network protocols employ bitmasking to specify flags and options that are used to modify the protocol's behavior.

• Advantages of Bitmasking:

1. **Greater speed:** Bit-level operations typically occur more quickly than more conventional operations like addition or multiplication. Bitmasking enables a speedy and effective execution of sophisticated operations on massive data sets.

2. **Memory efficiency:** Bitmasking can be used for compact and memory-efficient storage of big collections of data.

3. **Code simplicity:** By swapping out complex conditional statements with straightforward bit-level operations, bitmasking can make code simpler and more elegant.

4. **Versatility:** Bitmasking is a flexible method that can be applied to a variety of tasks, including network protocols, cryptography, and data compression.

5. **Data masking and filtering:** By selectively turning on or off specific bits, bitmasks can be used to mask or filter data.

6. **Bit-level programming:** For low-level programming activities like creating device drivers or operating systems, where it is important to directly manipulate bits and bytes, bitmasks are a crucial tool.

• **Disadvantages of Bitmasking:**

1. Limited range: Bitmasks have a limited range since they can only represent a finite range of values.

2. Code complexity: Bitmasking can make a program harder to comprehend, especially if it uses complicated bitwise operations.

3. Debugging: Bitwise operations have the potential to bring elusive bugs into programs. A misplaced bitwise operator can radically alter the meaning of an expression.

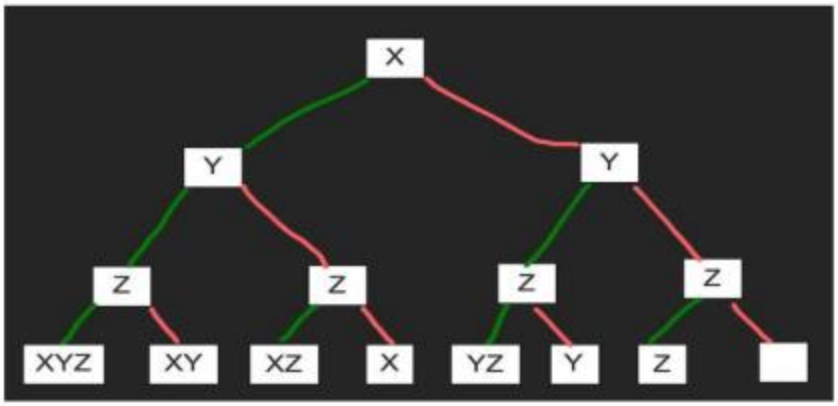
4. Restricted readability: While using bitmasks can make code shorter, those unfamiliar with the method may find it difficult to understand. As a result, maintaining and updating the code over time may be challenging.

• Finding all combinations

Let us say you are given a set S which has three elements S -> {X, Y, Z}. Now we want to find all possible sub-sets of the given set S.

1. Recursion

To solve this problem, we can see what are the possibilities associated with each element of the set. We find that there are only two possibilities – either you can include that element or you can exclude them. So for X there are two possibilities – I(Included), and E(Excluded), similarly for Y and Z there are two possibilities I, E. So, each element has two choices – the total number of choices will be 2 * 2 * 2 = 8. If we have n elements the number of possibilities will be 2n - 1.



The above diagram represents the recursion tree. When you are at X you have two options – include X or exclude it – green line represents inclusion, and the red represents exclusion. If you follow the tree diagram you will see that the bottom-most line represents the number of possible subsets. The range of a set containing n elements will be from an empty subset to a subset containing all the elements.

2. Bit Mask

In this section, we will see how to solve the above problem using a bit mask. We will use 0 to represent exclusion and 1 to represent inclusion. For example, if we have a set containing elements X, Y, Z we can create a mask 100. This will mean that X is included and Y and Z are excluded. So it means {X,Y,Z} -> 100 = {A}. Now if we want to exclude all the elements our mask should be 000, and if we want to include everything it should be 111.

	X	Y	Z	
0	0	0	0	{}
1	0	0	1	{Z}
2	0	1	0	{Y}
3	0	1	1	{Y, Z}
4	1	0	0	{X}
5	1	0	1	{X, Z}
6	1	1	0	{X, Y}
7	1	1	1	{X, Y, Z}

In the above diagram, the second row having 0 as the first element represents the combination where no element is included and the last row represents the combination where all the elements are included. Note that the range will be from 0 to 2n -1, in our case it will be 23 – 1 = 7. So when you are creating the bitmask for the given elements (say n), it includes running a loop from 0 to 2n -1.