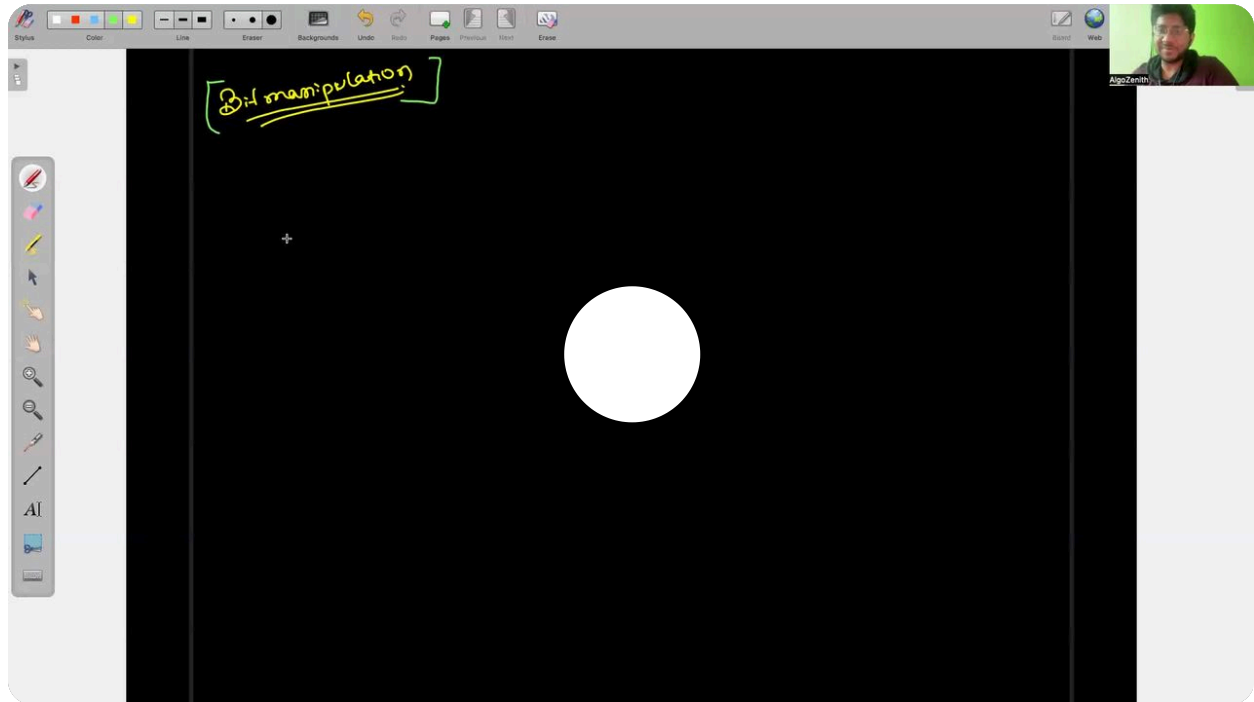




Bitmasking



Bit Manipulation

Bit manipulation involves the precise control and manipulation of individual bits in a binary number to achieve specific computational goals or perform operations efficiently. In the realm of computing, data is stored and processed in binary form, where the fundamental units are bits. These bits, denoted as 0s and 1s, constitute the binary language that forms the foundation of digital information processing.

Binary Representation: All data, including numbers, is internally represented in binary. In this representation, each digit is a bit, and the position of the bit contributes to the overall value. For instance, let's break down the binary number 1011:

- The rightmost bit (1) represents 2^0 , which is 1.
- Moving to the left, the next bit (1) represents 2^1 , which is 2.
- The third bit (0) represents 2^2 , which is 4.
- The leftmost bit (1) represents 2^3 , which is 8.

Now, to find the decimal equivalent, you sum up these values:

$$1 (2^0) + 2 (2^1) + 0 (2^2) + 8 (2^3) = 1 + 2 + 0 + 8 = 11$$

Now, let's go through each of the basic bitwise operations in detail:

A	B	A XOR B	A AND B	A OR B
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	0	1	1

Truth Table of bitwise operations

a. XOR (^ - Exclusive OR):

Performs a bitwise exclusive OR operation. The result is 1 if the corresponding bits are different; otherwise, it's 0.

Example:

```
A = 1010 (10)
B = 1100 (12)
A ^ B = 0110 (6)
```

[Copy](#)

b. Left Shift (<<):

Shifts the bits of a binary number to the left by a specified number of positions.

Example:

```
A = 1010 (10)
A << 2 = 101000 (40) (Shift A to the left by 2 positions)
```


[Copy](#)

c. Right Shift (>>):

Shifts the bits of a binary number to the right by a specified number of positions. For unsigned integers, zeros are shifted in from the left. For signed integers, the behavior is implementation-dependent.

Example:

```
A = 1010 (10)
A >> 1 = 0101 (5) (Right shift A by 1 position)
```


 Copy

d. NOT (~):

Inverts each bit of a binary number (flips 0s to 1s and vice versa).

Example:

```
A = 1010 (10)
~A = 0101 (5) (Bitwise NOT of A)
```


 Copy

e. AND (&):

Performs a bitwise AND operation. The result is 1 only if both corresponding bits are 1.

Example:

```
A = 1010 (10)
B = 1100 (12)
A & B = 1000 (8) (Bitwise AND of A and B)
```


 Copy

f. OR (|):

Performs a bitwise OR operation. The result is 1 if at least one of the corresponding bits is 1.

Example:

```
A = 1010 (10)
B = 1100 (12)
A | B = 1110 (14) (Bitwise OR of A and B)
```

 Copy

g. Subtraction (-):

In binary arithmetic, subtraction can be represented using the concept of two's complement. The two's complement is a way of representing signed integers in binary. The steps involved in finding the two's complement are:

1. Take the bitwise complement (NOT) of the subtrahend.

2. Add 1 to the result obtained in step 1.

Let's represent the subtraction of B from A using two's complement:

Example:

```
A = 1010 (10)
```

```
B = 1100 (12)
```

[Copy](#)

```
1. Take the bitwise complement (NOT) of B:
```

```
~B = 0011 (3)
```

```
2. Add 1 to the bitwise complement:
```

```
~B + 1 = 0100 (4)
```

```
Now, A - B is equivalent to A + (~B + 1):
```

```
A + (~B + 1) = 1010 + (0100) = 1010 + 0100 = 1110 (14)
```

In this example, subtraction of B from A is represented as the addition of A and the two's complement of B. The result is 1110 in binary, which is equivalent to the decimal result of subtracting B from A. The two's complement representation simplifies the subtraction process and allows for a consistent representation of signed integers in binary.

Operation (x - 1) and Flipping Bits

The operation (x - 1) in binary arithmetic has a specific effect: it flips all the bits to the right of the least significant set (1) bit in (x). Here's how it works:

1. Identify the Least Significant Set Bit (LSB):

- Find the rightmost bit that is set (1) in (x).

Example:

```
x = 1010100
```

```
LSB is quoted: 1010`1`00
```


[Copy](#)

2. Flip Bits to the Right of LSB:

- Flip (change 0s to 1s and 1s to 0s) all the bits to the right of the LSB (set) , including the LSB itself.

Example:

```
x - 1 = 1010100 - 1 = 1010011 (83)
```

 Copy

The bits to the right of the LSB (including the LSB) have been flipped.

This property is often used in bitwise operations to clear the rightmost set bit in a number or to find the next lower number that is a power of two.

Let's explore some important method which we use again and again:


`set(i)`, `unset(i)`, `check(i)`, `flip(i)`, and `count()` in the context of bitwise operations.

a. `set(i)`:

The operation `set(i)` is used to set the *i*th bit (counting from the right, starting with 0) in a binary number. It involves turning the *i*th bit from 0 to 1 while leaving other bits unchanged.

Example:

```
x = 10001 (17)
set(2) of x: 10001 | (1 << 2) = 10101 (21)
```

 Copy

In this example, setting the 2nd bit of `x` results in `10101`.

b. `unset(i)`:

The operation `unset(i)` is used to unset (clear) the *i*th bit in a binary number. It involves turning the *i*th bit from 1 to 0 while leaving other bits unchanged.

Example:

```
x = 1011100 (92)
unset(3) of x: 1011100 & ~(1 << 3) = 1010100 (84)
```

 Copy


In this example, unsetting the 3rd bit of `x` results in `1010100`.

c. `check(i)`:

The operation `check(i)` is used to check the value of the *i*th bit in a binary number. It returns true if the *i*th bit is set (1) and false if it is unset (0).

Example:

```
x = 1010100 (84)
check(4) of x: (x & (1 << 4)) != 0 // true
```

 Copy


In this example, checking the 4th bit of **x** returns true because the 4th bit is set.

d. flip(i):

The operation **flip(i)** is used to toggle the value of the *i*th bit in a binary number. It changes a set bit to unset and vice versa while leaving other bits unchanged.

Example:

```
x = 1010100 (84)
flip(1) of x: 1010100 ^ (1 << 1) = 1010101 (85)
```

 Copy


In this example, flipping the 1st bit of **x** results in **1010101**.

e. count():

The operation **count()** is used to count the number of set bits (1s) in a binary number. It involves iterating through each bit and counting the ones.

Example:

```
x = 1010100 (84)
count() of x: Count the set bits in 1010100 (84) = 3
```

 Copy

In this example, there are three set bits in the binary representation of **x**.

Understanding Bitmask

Bitmask refers to a binary number used to represent a subset of elements. Each bit in the bitmask corresponds to an element in the set, with a set bit indicating the presence of the corresponding element.

Representing a Set Using Bitmasking

When dealing with a set *S* comprising *N* items numbered from 1 to *N*, Bitmasking provides an effective way to represent every subset of *S* through binary numbers. In this

representation, each bit in the binary sequence corresponds to an item in the set, forming what is commonly known as a "mask." The binary sequence, or mask, serves as a compact representation of a subset of the set S .

Set Representation

The set S , denoted as $S = \{1, 2, 3, \dots, N\}$, can be represented using a binary sequence of N bits. Each bit in the sequence signifies the presence or absence of the corresponding item in the set.

For instance, the binary sequence "10101" serves as a mask, representing a subset of size 5. In this mask, if the i th bit is set to 1, it indicates that the i th item is included in the subset. Conversely, if the i th bit is unset or set to 0, the i th item is not part of the subset.

Example

Consider the mask "1010110." (86) In this example:

- The subset size is 7, corresponding to the number of bits in the mask.
- Items included in the subset are 1, 3, 5, and 6 (assuming 1-based indexing), where the corresponding bits are set to 1.

The following diagram illustrates the concept:

```
Mask:      1 0 1 0 1 1 0
Set:       1 2 3 4 5 6 7
Included:  ✓  ✓  ✓✓
```

This bitmask efficiently captures the composition of the subset, providing a concise representation for various set operations and algorithms.

Operations with Bitmask

Consider a set $A = \{1, 2, 3, 4, 5\}$. A bitmask of length 5 can represent any subset of A , where the i th bit being set implies the presence of the i th element. For instance, the bitmask 01010 represents the subset $\{2, 4\}$.

1. Set the i th Bit:

- Formula: $b \mid (1 \ll i)$
- Example: If the bitmask $b = 01010$, setting the 0th bit results in 01011, representing the subset $\{1, 2, 4\}$.

2. Unset the i th Bit:

- Formula: $b \& !(1 \ll i)$
- Example: If the bitmask $b = 01010$, unsetting the 1st bit results in 01000, representing the subset $\{4\}$.

3. Check if i th Bit is Set:


- Formula: $b \& (1 \ll i)$

- Example: If the bitmask $b = 01010$, checking if the 3rd bit is set results in a non-zero value, indicating the presence of the 3rd element in the subset.

Problem: Count Subsets with Sum Greater Than or Equal To a Given Value

Algorithm

```
solve(set, set_size, val)
    count = 0
    for x = 0 to power(2, set_size)
        sum = 0
        for k = 0 to set_size
            if kth bit is set in x
                sum = sum + set[k]
        if sum >= val
            count = count + 1
    return count
```

 Copy

- **Objective:** Count the number of subsets from a given set with a sum of elements greater than or equal to a specified value.
- **Complexity:** $O(2^n)$, where n is the size of the set.

Explanation

1. Iterate over all possible subsets represented by binary numbers from 0 to $2^{(\text{set_size})}-1$.
2. For each subset, calculate the sum of elements based on the set bits in the bitmask.
3. If the sum is greater than or equal to the given value, increment the count.
4. Return the final count.

Converting the algorithm into code and executing it will provide the desired result.

Note: This approach exploits the power of bitmask operations to efficiently handle subset generation and evaluation.

Let's see more usage of bitmasking.

Problem Statement: Given a set S of size N consisting of N items numbered from 1 to N . Print all the subsets of S . Set S can be represented as $S = \{1, 2, 3, \dots, N\}$. Constraints- N can range from any integer between 1 to 20 inclusive. i.e $1 \leq N \leq 20$.


Code:


```

void PrintAllSubsets(int N, int maximum_mask_required)
{
    //print the empty set or the null set seperately.
    cout << "0";

    //In the outer loop we will iterate through all the masks required
    // i.e from 0 to 2^N-1
    for (int mask = 0; mask <= maximum_mask_required; mask++)
    {
        //In the inner loop we will check whether the k-th bit of
        // the mask is set or not
        //If k-th bit is set k+1 th item is present in the subset
        // Else k+1-th item is not present in the current subset.
        for (int k = 0; k < N; k++)
        {
            if ((mask & (1 << k)) != 0)
            {
                // the k-th bit of the mask is set print k+1 (
                cout << k + 1 << " ";
            }
        }
        cout << "\n";
    }
}

```

 Copy

Explanation:

- 1. Set Representation:** Consider a set S with N elements.
- 2. Binary Representation:** Every subset of a set S can be represented using the binary representation of a number from 0 to 2^N-1 . This is because each bit in the binary representation corresponds to an element in the set (1 if the element is present, 0 if it's not).
- 3. Outer Loop:** The outer loop runs through all possible masks representing subsets of the set. It iterates from 0 to 2^N-1 .
- 4. Inner Loop:** In the inner loop, iterate through all the bits of a mask. Since there are N elements in the set, the inner loop runs from 0 to $N-1$. Each bit in the mask represents the presence or absence of an element in the current subset.
- 5. Check Bit Status:** If the i -th (0-based indexing) bit of the mask is set (1), it means that the $(i+1)$ -th (1-based indexing) item is present in the current subset. Print or process the item accordingly. If the bit is not set (0), continue the loop.

6. Subset Printing/Processing: At the end of the outer loop, you will have iterated through all possible subsets of the given set S . You can print or process each subset as needed.

Time Complexity:

To generate all the subsets of a set S having N elements, we use a two-loop approach. The outer loop runs from 0 to $(2^N - 1)$, iterating through all the masks required to represent all subsets of size N . In the inner loop, we iterate through each bit in the mask. Since there are N items in the set S , we iterate through N bits in the inner loop from 0 to $(N - 1)$.

Thus, the time complexity for the given approach is $(O(2^N * N))$.

Problem Statement: Given a set S of size N consisting of N items. Print the all subsets of the subsets. Ex: $a[n] = \{1, 2\}$

Output: Subset 0: $\{\}$ Subset 1: $\{1\}$ Subset 2: $\{2\}$ Subset 3: $\{1\ 2\}$ $\{2\}$ $\{1\}$

```
#include <iostream>
using namespace std;

int main()
{
    int n;

    std::cin >> n;


    int arr[n];

    for (int i = 0; i < n; i++)
    {
        std::cin >> arr[i];
    }

    for (int mask = 0; mask < (1 << n); mask++)
    {
        cout << "Subset " << mask << ": ";

        // Iterate through submasks
        for (int submask = mask; submask; submask = (submask - 1) & mask)
        {
            cout << "{ ";

            // Iterate through positions
            for (int pos = 0; pos < n; pos++)
            {
```

 Copy

```
        if ((submask & (1 << pos)) != 0)
        {
            cout << arr[pos] << " ";
        }

        cout << "} ";
    }

    cout << endl;
}

return 0;
}
```

Explanation:

- **Initialization:**

- The loop starts by initializing `submask` with the value of `mask`.

- **Condition Check:**

- The loop continues iterating as long as `submask` is non-zero.
- It terminates when `submask` becomes zero.

- **Update Expression:**

- In each iteration, `submask` is updated using the expression `(submask - 1) & mask`.
- `(submask - 1)` subtracts 1 from `submask`, effectively flipping the rightmost set bit to 0.
- `& mask` performs a bitwise AND operation with `mask`, ensuring that only the bits set in the original `mask` are considered.

- **Termination:**

- The loop effectively iterates through all possible submasks of the original `mask`.
- It terminates when there are no more set bits in `submask`.

Time Complexity:

The time complexity of the given code is $O(2^n \cdot n)$, where n is the size of the array `arr`.

Let's break down the time complexity analysis:

1. **Outer Loop:** The outer loop runs for 2^n iterations, where n is the size of the array. This is because the loop iterates through all possible subsets of the array `arr`, and there are 2^n subsets for an array of size n .

2. **Inner Loop (Submask Iteration):** The inner loop is based on the generation of submasks using the expression `(submask - 1) & mask`. In the worst case, this loop iterates $O(3^n)$ times. This is because, for each subset represented by the `mask`, the inner loop iterates through all possible submasks.
3. **Nested Iteration:** Within the innermost loop (position iteration), the code iterates through each position in the array, which takes $O(n)$ time.

Bitmask Manipulation FAQs

What is an advanced use case for bitmask manipulation?

Advanced bitmask manipulation is often employed in scenarios where compactly representing and efficiently manipulating complex sets of binary flags or configurations is crucial. This includes optimizing algorithms, compressing data structures, or managing intricate state machines.

Can I use bitmask manipulation for range queries?

Yes, bitmask manipulation can be extended for range queries. You can create a bitmask that represents a range of bits and perform operations within that range. Shifting and masking appropriately allows you to manipulate and extract information from the selected range.

How do I efficiently toggle a range of bits?

To toggle a range of bits, create a bitmask that covers the range and use the XOR operator (`^`). For instance, to toggle bits 2 through 5, use `num ^= (0b1111 << 2)`.

Is there a technique to set or clear bits based on a condition?

Yes, you can use conditional statements with the bitwise OR (`|`) and AND (`&`) operators. For example, to set the 5th bit only if the 2nd bit is already set, use `num |= ((num & (1 << 1)) ? (1 << 4) : 0)`.

How can I efficiently find the position of the rightmost set bit?

To find the position of the rightmost set bit, use the bitwise AND (`&`) with the two's complement of the number (`num & -num`). The result will have only the rightmost set bit of the original number.

What are some pitfalls to watch out for in advanced bitmask manipulation?

Be cautious when dealing with signed integers, as right shifts on negative numbers might behave differently depending on the programming language and platform. Additionally, ensure that your bitwise operations are endian-independent for cross-platform compatibility.