Bitwise Algorithms

[**Learn more about Bitwise Algorithms in DSA Self Paced Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=geeksforgeeks&utm_medium=articles+bit_algo_lp+header_link_click&utm_campaign=dsa+course+tracker)

[**Practice Problems on Bit Magic !**](https://practice.geeksforgeeks.org/explore/?category%5B%5D=Bit%20Magic&page=1&category%5B%5D=Bit%20Magic&utm_source=geeksforgeeks&utm_medium=articles+bit_algo_lp+header_link_click&utm_campaign=practice+tracker)

[**Recent Articles on Bit Magic !**](https://www.geeksforgeeks.org/category/bit-magic/)

The **Bitwise Algorithms** is used to perform operations at the bit-level or to manipulate bits in different ways. The bitwise operations are found to be much faster and are sometimes used to improve the efficiency of a program.

**For example**: To check if a number is even or odd. This can be easily done by using Bitwise-AND(&) operator. If the last bit of the operator is set than it is ODD otherwise it is EVEN. Therefore, if **num & 1** not equals to zero than num is ODD otherwise it is EVEN.

**Topics:**

* [Introduction](https://www.geeksforgeeks.org/bitwise-algorithms/#introduction)
* [Standard Problems on Bit Algorithms](https://www.geeksforgeeks.org/bitwise-algorithms/#standard)

**Introduction:**

1. [Introduction to Bitwise Algorithms – Data Structures and Algorithms Tutorial](https://www.geeksforgeeks.org/introduction-to-bitwise-algorithms-data-structures-and-algorithms-tutorial/)
2. [Bitwise Operators in C/C++](https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/)
3. [Bitwise Operators in Java](https://www.geeksforgeeks.org/bitwise-operators-in-java/)
4. [Python Bitwise Operators](https://www.geeksforgeeks.org/python-bitwise-operators/)
5. [JavaScript Bitwise Operators](https://www.geeksforgeeks.org/javascript-bitwise-operators/)
6. [All about Bit Manipulation](https://www.geeksforgeeks.org/all-about-bit-manipulation/)
7. [Little and Big Endian Mystery](https://www.geeksforgeeks.org/little-and-big-endian-mystery/)

**Standard Problems on Bit Algorithms:**

* **Easy:**
  1. [Binary representation of a given number](https://www.geeksforgeeks.org/binary-representation-of-a-given-number/)
  2. [Count set bits in an integer](https://www.geeksforgeeks.org/count-set-bits-in-an-integer/)
  3. [Add two bit strings](https://www.geeksforgeeks.org/add-two-bit-strings/)
  4. [Turn off the rightmost set bit](https://www.geeksforgeeks.org/turn-off-the-rightmost-set-bit/)
  5. [Rotate bits of a number](https://www.geeksforgeeks.org/rotate-bits-of-an-integer/)
  6. [Compute modulus division by a power-of-2-number](https://www.geeksforgeeks.org/compute-modulus-division-by-a-power-of-2-number/)
  7. [Find the Number Occurring Odd Number of Times](https://www.geeksforgeeks.org/find-the-number-occurring-odd-number-of-times/)
  8. [Program to find whether a no is power of two](https://www.geeksforgeeks.org/program-to-find-whether-a-no-is-power-of-two/)
  9. [Find position of the only set bit](https://www.geeksforgeeks.org/find-position-of-the-only-set-bit/)
  10. [Check for Integer Overflow](https://www.geeksforgeeks.org/check-for-integer-overflow/)
  11. [Find XOR of two number without using XOR operator](https://www.geeksforgeeks.org/find-xor-of-two-number-without-using-xor-operator/)
  12. [Check if two numbers are equal without using arithmetic and comparison operators](https://www.geeksforgeeks.org/check-if-two-numbers-are-equal-without-using-arithmetic-and-comparison-operators/)
  13. [Detect if two integers have opposite signs](https://www.geeksforgeeks.org/detect-if-two-integers-have-opposite-signs/)
  14. [How to swap two numbers without using a temporary variable?](https://www.geeksforgeeks.org/swap-two-numbers-without-using-temporary-variable/)
  15. [Russian Peasant (Multiply two numbers using bitwise operators)](https://www.geeksforgeeks.org/russian-peasant-multiply-two-numbers-using-bitwise-operators/)
* **Medium:**
  1. [Swap bits in a given number](https://www.geeksforgeeks.org/swap-bits-in-a-given-number/)
  2. [Smallest of three integers without comparison operators](https://www.geeksforgeeks.org/smallest-of-three-integers-without-comparison-operators/)
  3. [Compute the minimum or maximum of two integers without branching](https://www.geeksforgeeks.org/compute-the-minimum-or-maximum-max-of-two-integers-without-branching/)
  4. [Smallest power of 2 greater than or equal to n](https://www.geeksforgeeks.org/smallest-power-of-2-greater-than-or-equal-to-n/)
  5. [Write a C program to find the parity of an unsigned integer](https://www.geeksforgeeks.org/write-a-c-program-to-find-the-parity-of-an-unsigned-integer/)
  6. [Check if binary representation of a number is palindrome](https://www.geeksforgeeks.org/check-binary-representation-number-palindrome/)
  7. [Generate n-bit Gray Codes](https://www.geeksforgeeks.org/given-a-number-n-generate-bit-patterns-from-0-to-2n-1-so-that-successive-patterns-differ-by-one-bit/)
  8. [Check if a given number is sparse or not](https://www.geeksforgeeks.org/check-if-a-given-number-is-sparse-or-not/)
  9. [Euclid’s Algorithm when % and / operations are costly](https://www.geeksforgeeks.org/euclids-algorithm-when-and-operations-are-costly/)
  10. [Calculate square of a number without using \*, / and pow()](https://www.geeksforgeeks.org/calculate-square-of-a-number-without-using-and-pow/)
  11. [Cyclic Redundancy Check and Modulo-2 Division](https://www.geeksforgeeks.org/modulo-2-binary-division/)
  12. [Copy set bits in a range](https://www.geeksforgeeks.org/copy-set-bits-in-a-range/)
  13. [Check if a number is Bleak](https://www.geeksforgeeks.org/check-if-a-number-is-bleak/)
  14. [Gray to Binary and Binary to Gray conversion](https://www.geeksforgeeks.org/gray-to-binary-and-binary-to-gray-conversion/)
* **Hard:**
  1. [Next higher number with same number of set bits](https://www.geeksforgeeks.org/next-higher-number-with-same-number-of-set-bits/)
  2. [Karatsuba algorithm for fast multiplication](https://www.geeksforgeeks.org/divide-and-conquer-set-2-karatsuba-algorithm-for-fast-multiplication/)
  3. [Find the maximum subarray XOR in a given array](https://www.geeksforgeeks.org/find-the-maximum-subarray-xor-in-a-given-array/)
  4. [Find longest sequence of 1’s in binary representation with one flip](https://www.geeksforgeeks.org/find-longest-sequence-1s-binary-representation-one-flip/)
  5. [Closest (or Next) smaller and greater numbers with same number of set bits](https://www.geeksforgeeks.org/closest-next-smaller-greater-numbers-number-set-bits/)
  6. [Bitmasking and Dynamic Programming | Set-2 (TSP)](https://www.geeksforgeeks.org/bitmasking-dynamic-programming-set-2-tsp/)
  7. [Compute the parity of a number using XOR and table look-up](https://www.geeksforgeeks.org/compute-parity-number-using-xor-table-look/)
  8. [XOR Encryption by Shifting Plaintext](https://www.geeksforgeeks.org/xor-encryption-shifting-plaintext/)
  9. [Count pairs in an array which have at least one digit common](https://www.geeksforgeeks.org/count-pairs-array-least-one-digit-common/)
  10. [Python program to convert floating to binary](https://www.geeksforgeeks.org/python-program-to-convert-floating-to-binary/)
  11. [Booth’s Multiplication Algorithm](https://www.geeksforgeeks.org/booths-multiplication-algorithm/)
  12. [Number of pairs with Pandigital Concatenation](https://www.geeksforgeeks.org/number-pairs-pandigital-concatenation/)
  13. [Find the n-th number whose binary representation is a palindrome](https://www.geeksforgeeks.org/find-n-th-number-whose-binary-representation-palindrome/)
  14. [Find the two non-repeating elements in an array of repeating elements](https://www.geeksforgeeks.org/find-two-non-repeating-elements-in-an-array-of-repeating-elements/)

**Binary representation of a given number**

Write a program to print Binary representation of a given number.

**Method 1: Iterative**

For any number, we can check whether its ‘i’th bit is 0(OFF) or 1(ON) by bitwise ANDing it with “2^i” (2 raise to i).

1) Let us take number 'NUM' and we want to check whether it's 0th bit is ON or OFF   
 bit = 2 ^ 0 (0th bit)  
 if NUM & bit >= 1 means 0th bit is ON else 0th bit is OFF

2) Similarly if we want to check whether 5th bit is ON or OFF   
 bit = 2 ^ 5 (5th bit)  
 if NUM & bit >= 1 means its 5th bit is ON else 5th bit is OFF.

Let us take unsigned integer (32 bit), which consist of 0-31 bits. To print binary representation of unsigned integer, start from 31th bit, check whether 31th bit is ON or OFF, if it is ON print “1” else print “0”. Now check whether 30th bit is ON or OFF, if it is ON print “1” else print “0”, do this for all bits from 31 to 0, finally we will get binary representation of number.

**def** bin(n) :

    i **=** 1 << 31

**while**(i > 0) :

**if**((n & i) !**=** 0) :

**print**("1", end **=** "")

**else** :

**print**("0", end **=** "")

        i **=** i **//** 2

bin(7)

print()

bin(4)

# This code is contributed by divyeshrabadiya07.

**Output**

00000000000000000000000000000111  
00000000000000000000000000000100

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

**Method 2: Recursive**

Following is recursive method to print binary representation of ‘NUM’.

step 1) if NUM > 1  
 a) push NUM on stack  
 b) recursively call function with 'NUM / 2'  
step 2)  
 a) pop NUM from stack, divide it by 2 and print it's remainder.

# Python3 Program for the binary

# representation of a given number

**def** bin(n):

**if** n > 1:

        bin(n**//**2)

**print**(n **%** 2, end**=**"")

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    bin(7)

    print()

    bin(4)

# This code is contributed by ANKITRAI1

**Output**

111  
100

**Time Complexity:**O(log N)

**Auxiliary Space:**O(log N)

**Method 3: Recursive using bitwise operator**

Steps to convert decimal number to its binary representation are given below:

step 1: Check n > 0  
step 2: Right shift the number by 1 bit and recursive function call  
step 3: Print the bits of number

# Python 3 implementation of above approach

# Function to convert decimal to

# binary number

**def** bin(n):

**if** (n > 1):

        bin(n >> 1)

**print**(n & 1, end**=**"")

# Driver code

bin(131)

print()

bin(3)

# This code is contributed by PrinciRaj1992

**Output**

10000011  
11

**Time Complexity:**O(log N)

**Auxiliary Space:** O(log N)

**Method 4:** **Using Bitset of C++**

We can use the*bitset class of C++* to store the binary representation of any number (positive as well as a negative number). It offers us the flexibility to have the number of bits of our desire, like whether we want to have 32-bit binary representation of just an 8-bit representation of a number.

A complete guide to using bitset can be found on this gfg article [**LINK**](https://www.geeksforgeeks.org/c-bitset-and-its-application/).

# Importing the necessary module

**import** sys

# Initialize two variables with values 5 and -5

n **=** 5

m **= -**5

# Using the format function to convert integers to binary

b **=** bin(n & int("1"**\***8, 2))[2:].zfill(8)

b1 **=** bin(m & int("1"**\***8, 2))[2:].zfill(8)

# Printing the binary representation of n and m

print("Binary of 5:", b)

**print**("Binary of -5:", b1)

# This code is contributed by Vikram\_Shirsat

**Output**

Binary of 5:00000101  
Binary of -5:11111011

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

**Method 5: Using Inbuilt library:**

**def** binary(num):

**return** int(bin(num).split('0b')[1])

**if** \_\_name\_\_ **==** "\_\_main\_\_" :

    x **=** 10

    binary\_x **=** binary(x)

    print("the binary number is :",binary\_x)

# This code is contributed by Rishika Gupta.

**Output**

1010

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

**Count set bits in an integer**

Write an efficient program to count the number of 1s in the binary representation of an integer.

**Examples :**

***Input :****n = 6*

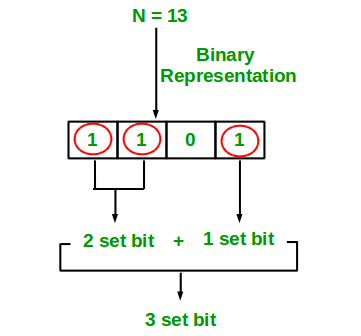
***Output :****2*

*Binary representation of 6 is 110 and has 2 set bits*

***Input :****n = 13*

***Output :****3*

*Binary representation of 13 is 1101 and has 3 set bits*



Number of 1 Bits

**1. Simple Method** Loop through all bits in an integer, check if a bit is set and if it is, then increment the set bit count. See the program below.

# Python3 program to Count set

# bits in an integer

# Function to get no of set bits in binary

# representation of positive integer n \*/

**def**  countSetBits(n):

    count **=** 0

**while** (n):

        count **+=** n & 1

        n >>**=** 1

**return** count

# Program to test function countSetBits \*/

i **=** 9

print(countSetBits(i))

# This code is contributed by

# Smitha Dinesh Semwal

**Output**

2

**Time Complexity:** O(log n)

**Auxiliary Space:**O(1)

**Recursive Approach:**

# Python3 implementation of recursive

# approach to find the number of set

# bits in binary representation of

# positive integer n

**def** countSetBits( n):

    # base case

**if** (n **==** 0):

**return** 0

**else**:

        # if last bit set add 1 else

        # add 0

**return** (n & 1) **+** countSetBits(n >> 1)

# Get value from user

n **=** 9

# Function calling

**print**( countSetBits(n))

# This code is contributed by sunnysingh

**Output**

2

**Time Complexity:** O(log n)

**Auxiliary Space:**O(log n)for recursive stack space

**2. Brian Kernighan’s Algorithm:**

Subtracting 1 from a decimal number flips all the bits after the rightmost set bit(which is 1) including the rightmost set bit.

for example :

10 in binary is 00001010

9 in binary is 00001001

8 in binary is 00001000

7 in binary is 00000111

So if we subtract a number by 1 and do it bitwise & with itself (n & (n-1)), we unset the rightmost set bit. If we do n & (n-1) in a loop and count the number of times the loop executes, we get the set bit count.

The beauty of this solution is the number of times it loops is equal to the number of set bits in a given integer.

1 Initialize count: = 0  
 2 **If** integer n is not zero  
 (a) Do bitwise & with (n-1) and assign the value back to n  
 n: = n&(n-1)  
 (b) Increment count by 1  
 (c) go to step 2  
 3 **Else** return count

**Example for Brian Kernighan’s Algorithm:**

n = 9 (1001)  
 count = 0

Since 9 > 0, subtract by 1 and do bitwise & with (9-1)  
 n = 9&8 (1001 & 1000)  
 n = 8  
 count = 1

Since 8 > 0, subtract by 1 and do bitwise & with (8-1)  
 n = 8&7 (1000 & 0111)  
 n = 0  
 count = 2

Since n = 0, return count which is 2 now.

**Implementation of Brian Kernighan’s Algorithm:**

# Function to get no of set bits in binary

# representation of passed binary no. \*/

**def** countSetBits(n):

    count **=** 0

**while** (n):

        n &**=** (n**-**1)

        count**+=** 1

**return** count

# Program to test function countSetBits

i **=** 9

print(countSetBits(i))

# This code is contributed by

# Smitha Dinesh Semwal

**Output**

2

**Time Complexity:** O(log n)

**Auxiliary Space:** O(1)

**Recursive Approach:**

# Python3 implementation for

# recursive approach to find

# the number of set bits using

# Brian Kernighan’s Algorithm

# recursive function to count

# set bits

**def** countSetBits(n):

    # base case

**if** (n **==** 0):

**return** 0

**else**:

**return** 1 **+** countSetBits(n & (n **-** 1))

# Get value from user

n **=** 9

# function calling

print(countSetBits(n))

# This code is contributed by sunnysingh

**Output**

2

**Time Complexity:** O(log n)

**Auxiliary Space:** O(log n)

**3. Using Lookup table:**We can count bits in O(1) time using the lookup table.

Below is the implementation of the above approach:

# Python implementation of the approach

BitsSetTable256 **=** [0] **\*** 256

# Function to initialise the lookup table

**def** initialize():

    # To initially generate the

    # table algorithmically

    BitsSetTable256[0] **=** 0

**for** i **in** range(256):

        BitsSetTable256[i] **=** (i & 1) **+** BitsSetTable256[i **//** 2]

# Function to return the count

# of set bits in n

**def** countSetBits(n):

**return** (BitsSetTable256[n & 0xff] **+**

            BitsSetTable256[(n >> 8) & 0xff] **+**

            BitsSetTable256[(n >> 16) & 0xff] **+**

            BitsSetTable256[n >> 24])

# Driver code

# Initialise the lookup table

initialize()

n **=** 9

print(countSetBits(n))

# This code is contributed by SHUBHAMSINGH10

**Output**

2

**Time Complexity:** O(1)

**Auxiliary Space:**O(1)

We can find one use of counting set bits at [Count number of bits to be flipped to convert A to B](https://www.geeksforgeeks.org/count-number-of-bits-to-be-flipped-to-convert-a-to-b/)

**Note:** In GCC, we can directly count set bits using \_\_builtin\_popcount(). So we can avoid a separate function for counting set bits.

# Python3 program to demonstrate \_\_builtin\_popcount()

**print**(bin(4).count('1'));

print(bin(15).count('1'));

# This code is Contributed by mits

**Output**

1  
4

**Time complexity:**O(1)

**Auxiliary space:**O(1)

**4. Mapping numbers with the bit.** It simply maintains a map(or array) of numbers to bits for a nibble. A Nibble contains 4 bits. So we need an array of up to 15.

int num\_to\_bits[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};

Now we just need to get nibbles of a given long/int/word etc recursively.

# Python3 program to count set bits by pre-storing

# count set bits in nibbles.

num\_to\_bits **=**[0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4];

# Recursively get nibble of a given number

# and map them in the array

**def** countSetBitsRec(num):

    nibble **=** 0;

**if**(0 **==** num):

**return** num\_to\_bits[0];

    # Find last nibble

    nibble **=** num & 0xf;

    # Use pre-stored values to find count

    # in last nibble plus recursively add

    # remaining nibbles.

**return** num\_to\_bits[nibble] **+** countSetBitsRec(num >> 4);

# Driver code

num **=** 31;

print(countSetBitsRec(num));

# this code is contributed by mits

**Output**

5

**Time Complexity:** O(log n), because we have log(16, n) levels of recursion.

**Storage Complexity:** O(1) Whether the given number is short, int, long, or long long we require an array of 16 sizes only, which is constant.

**5. Checking each bit in a number:**

Each bit in the number is checked for whether it is set or not. The number is bitwise AND with powers of 2, so if the result is not equal to zero, we come to know that the particular bit in the position is set.

# Check each bit in a number is set or not

# and return the total count of the set bits

**def** countSetBits(N):

    count **=** 0

    # (1 << i) = pow(2, i)

**for** i **in** range(4**\***8):

**if**(N & (1 << i)):

            count **+=** 1

**return** count

    # Driver code

N **=** 15

print(countSetBits(N))

# This code is contributed by avanitrachhadiya2155

**Output**

4

**Time complexity**: O(1)

**Auxiliary space**: O(1)

**6. Using power of 2:(efficient method to find for large value also)**

Iterate from k to 0 , where k is the largest power of 2 such that pow(2, k) <= num . And check if the Bitwise AND of num and pow(2, i) is greater than zero or not. If it is greater than zero , Then i-th bit is set ,then increase the count by 1.

# Python implementation of the above approach

**import** math

# Function to find largest power of 2 such that

# pow(2,k) <= N

**def** findk(n):

    i **=** 0

    val **=** math.pow(2, i)

**while** val <**=** n:

        k **=** i

        i **+=** 1

        val **=** math.pow(2, i)

**return** k

# Function to count set bits in a number

**def** countSetBits(N):

    count **=** 0

    k **=** findk(N)

**for** i **in** range(k, **-**1, **-**1):

        val **=** int(math.pow(2, i))

        x **=** val & N # x will store Bitwise AND of N & val

**if** x > 0:

            count **+=** 1

**return** count

# Drive Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    N **=** 15

    # Function call

    print(countSetBits(N))

**Output**

4

**Time Complexity:**O(logn)

**Auxiliary Space:** O(1)

**Add two bit strings**

Given two bit sequences as strings, write a function to return the addition of the two sequences. Bit strings can be of different lengths also. For example, if string 1 is “1100011” and second string 2 is “10”, then the function should return “1100101”.

[We strongly recommend that you click here and practice it, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/add-binary-strings3805/1)

Since the sizes of two strings may be different, we first make the size of a smaller string equal to that of the bigger string by adding leading 0s. After making sizes the same, we one by one add bits from rightmost bit to leftmost bit. In every iteration, we need to sum 3 bits: 2 bits of 2 given strings and carry. The sum bit will be 1 if, either all of the 3 bits are set or one of them is set. So we can do XOR of all bits to find the sum bit. How to find carry – carry will be 1 if any of the two bits is set. So we can find carry by taking OR of all pairs. Following is step by step algorithm.

**1.** Make them equal sized by adding 0s at the beginning of smaller string.

**2.**Perform bit addition

…..Boolean expression for adding 3 bits a, b, c

…..Sum = a XOR b XOR c

…..Carry = (a AND b) OR ( b AND c ) OR ( c AND a )

Following is implementation of the above algorithm.

# Python3 program for above approach

# adds the two-bit strings and return the result

# Helper method: given two unequal sized bit strings,

# converts them to same length by adding leading 0s

# in the smaller string. Returns the new length

**def** makeEqualLength(str1, str2):

    len1 **=** len(str1)     # Length of string 1

    len2 **=** len(str2)     # length of string 2

**if** len1 < len2:

        str1 **=** (len2 **-** len1) **\*** '0' **+** str1

        len1 **=** len2

**elif** len2 < len1:

        str2 **=** (len1 **-** len2) **\*** '0' **+** str2

        len2 **=** len1

**return** len1, str1, str2

**def** addBitStrings( first, second ):

    result **=** '' # To store the sum bits

    # make the lengths same before adding

    length, first, second **=** makeEqualLength(first, second)

    carry **=** 0 # initialize carry as 0

    # Add all bits one by one

**for** i **in** range(length **-** 1, **-**1, **-**1):

        firstBit **=** int(first[i])

        secondBit **=** int(second[i])

        # boolean expression for sum of 3 bits

        sum **=** (firstBit ^ secondBit ^ carry) **+** 48

        result **=** chr(sum) **+** result

        # boolean expression for 3 bits addition

        carry **=** (firstBit & secondBit) | \

                (secondBit & carry) | \

                (firstBit & carry)

        # if overflow, then add a leading 1

**if** carry **==** 1:

        result **=** '1' **+** result

**return** result

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    str1 **=** '1100011'

    str2 **=** '10'

    print('Sum is', addBitStrings(str1, str2))

# This code is contributed by

# chaudhary\_19 (Mayank Chaudhary)

**Output**

Sum is 1100101

***Time Complexity:****O(|str1|)*

***Auxiliary Space:****O(1)*

**Method – 2 (without adding extra zeros(0) in beginning of a small length string to make both strings with same length)**

Algo :

1. make to pointer i,j and set i = str1.size() – 1 and j = str2.size() – 1
2. take initial carry as 0 ans ans string as empty (“”)
3. while i>=0 or j>=0 or carry
4. add value of str1[i] and str2[j] in carry
5. add (carry%2) in resulting(answer string) string
6. set carry = carry/2
7. return ans

# The function that adds two-bit sequences and returns the addition

**def** addBitStrings(str1, str2):

    ans **=** ''

    i **=** len(str1) **-** 1

    j **=** len(str2) **-** 1

    carry **=** 0

**while** i >**=** 0 **or** j >**=** 0 **or** carry:

**if** i >**=** 0:

            carry **+=** ord(str1[i]) **-** ord('0')

            i **=** i **-** 1

**else**:

            carry **+=** 0

**if** j >**=** 0:

            carry **+=** ord(str2[j]) **-** ord('0')

            j **=** j **-** 1

**else**:

            carry **+=** 0

        ans **=** chr(ord('0') **+** carry **%** 2) **+** ans

        carry **=** carry **//** 2

**return** ans

# Driver program to test above functions

str1 **=** '1100011'

str2 **=** '10'

print('Sum is ', addBitStrings(str1, str2))

# This code is contributed by ajaymakavan.

**Output**

Sum is 1100101

Time Complexity: O(max(n,m)) (where, n = sizeof str1 & m = sizeof str2)  
Space Complexity: O(1)

**Turn off the rightmost set bit**

Write a program that unsets the rightmost set bit of an integer.

**Examples :**

Input: 12 (00...01100)  
Output: 8 (00...01000)

Input: 7 (00...00111)  
Output: 6 (00...00110)

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

Let the input number be n. n-1 would have all the bits flipped after the rightmost set bit (including the set bit). So, doing n&(n-1) would give us the required result.

So, now let us see how **n – 1** is **flipping all the bits** to the **right** (**including** the **rightmost set bit** also) of the **n**.

Taking n = 12, so (n – 1) = 11,

**n** can be written like **(n = (n – 1) + 1)**, so now we can think of this problem as[**Adding 1 to Any Number**](https://www.geeksforgeeks.org/add-1-to-a-given-number/) **(refer this article for better understanding)**

[**Binary representation**](https://www.geeksforgeeks.org/binary-representation-of-a-given-number/) of (n-1) = 11 = 1011, so now lets make n from (n-1), which can be done by adding a 1 to (n-1)

*On adding 1 to any number X, all bits to the right of****rightmost 0****(****including****the****rightmost zero****) gets****flipped***

*(n-1) = 1011*

(n-1) + 1 = 1100 ***(all the bits to the right of rightmost zero (including rightmost zero) got flipped)***

Since we have flipped the **rightmost zero**, so now, ***rightmost zero is now flipped to rightmost 1 (aka the rightmost set bit of n)*** and **all bits before rightmost 0 are the same as before**

X = 010 . . . . . 0 (rightmost zero) 111

X + 1 = 010 . . . . . 1 (rightmost one) 0 0 0

Example :

**X = 71 = Think of it as n – 1**

Binary Representation of X = 1000111

**X + 1 = 72 = Think of it as n**

Binary Representation of (X+1) = 1001000

**Observation :**

***1. All the bits to the left of rightmost 0 (excluding rightmost 0) in X (thinking it as n – 1) are same as in to the left of the rightmost 1(excluding rightmost 1)  in X + 1 (thinking*of *it as n)***

***2. All the bits to the right of rightmost 0 (including rightmost 0) in X (thinking it as n – 1) are different as in to the right of the rightmost 1 (including rightmost 1) in X + 1 (thinking*of *it as n)***

So **bitwise AND of left part of X**(till rightmost 0, excluding rightmost 0) and **left part of X + 1** (till rightmost 1, excluding rightmost 1) **will give the required answer**,  **bitwise AND right part of X** (from rightmost 0) and **right part of X + 1** **(from rightmost 1 (rightmost set bit))** will result in **0**

# unsets the rightmost set bit

# of n and returns the result

**def** fun(n):

**return** n & (n**-**1)

# Driver code

n **=** 7

print("The number after unsetting the rightmost set bit", fun(n))

# This code is contributed

# by Anant Agarwal.

**Output**

The number after unsetting the rightmost set bit 6

***Time Complexity:****O(1)*

***Auxiliary Space:****O(1)*

**Another Approach:**

The rightmost set bit can be switched off by subtracting the LSB from the number.

The LSB of a number can be obtained using (n & (-n)), therefore the number with the rightmost set bit of n switched off is equal to n – (n & (-n));

# Python3 program to implement the approach

# unsets the rightmost set bit

# of n and returns the result

**def** fun(n):

**return** n **-** (n & (**-**n))

# Driver Code

n **=** 7

print("The number after unsetting the rightmost set bit:", fun(n))

# This code is contributed by phasing17

**Output**

The number after unsetting the rightmost set bit: 6

**Time Complexity**: O(1)

**Auxiliary Space**: O(1)

**Rotate bits of a number**

Bit Rotation: A rotation (or circular shift) is an operation similar to shift except that the bits that fall off at one end are put back to the other end.

In left rotation, the bits that fall off at left end are put back at right end.

In right rotation, the bits that fall off at right end are put back at left end.

Rotate Bits

Example:

Let n is stored using 8 bits. Left rotation of n = 11100101 by 3 makes n = 00101111 (Left shifted by 3 and first 3 bits are put back in last ). If n is stored using 16 bits or 32 bits then left rotation of n (000…11100101) becomes 00..00**11100101**000.

Right rotation of n = 11100101 by 3 makes n = 10111100 (Right shifted by 3 and last 3 bits are put back in first ) if n is stored using 8 bits. If n is stored using 16 bits or 32 bits then right rotation of n (000…11100101) by 3 becomes **101**000..00**11100**.

# Python3 code to

# rotate bits of number

INT\_BITS **=** 32

# Function to left

# rotate n by d bits

**def** leftRotate(n, d):

    # In n<<d, last d bits are 0.

    # To put first 3 bits of n at

    # last, do bitwise or of n<<d

    # with n >>(INT\_BITS - d)

**return** (n << d)|(n >> (INT\_BITS **-** d))

# Function to right

# rotate n by d bits

**def** rightRotate(n, d):

    # In n>>d, first d bits are 0.

    # To put last 3 bits of at

    # first, do bitwise or of n>>d

    # with n <<(INT\_BITS - d)

**return** (n >> d)|(n << (INT\_BITS **-** d)) & 0xFFFFFFFF

# Driver program to

# test above functions

n **=** 16

d **=** 2

**print**("Left Rotation of",n,"by"

      ,d,"is",end**=**" ")

print(leftRotate(n, d))

print("Right Rotation of",n,"by"

     ,d,"is",end**=**" ")

print(rightRotate(n, d))

# This code is contributed by

# Smitha Dinesh Semwal

**Output**

Left Rotation of 16 by 2 is 64  
Right Rotation of 16 by 2 is 4

***Time Complexity:****O(1)*

***Auxiliary Space:****O(1)*

**For 16 bit number:**

SHORT\_SIZE **=** 16

# function to rotate the given unsigned short

# in the left direction

**def** leftRotate(x, d):

**return** (x << d) | (x >> (SHORT\_SIZE **-** d))

  # function to rotate the given unsigned short

# in the right direction

**def** rightRotate(x, d):

**return** (x >> d) | (x << (SHORT\_SIZE **-** d)) & 0xDDDDDF

# Driver program to test above functions

# Test case

n **=** 28

d **=** 2

**print**("Left Rotation of",n,"by"

      ,d,"is",end**=**" ")

print(leftRotate(n, d))

**print**("Right Rotation of",n,"by"

     ,d,"is",end**=**" ")

print(rightRotate(n, d))

# This code is contributed by shivanisinghss2110

**Output**

112  
7

**Time Complexity**: O(1)

**Space Complexity**: O(1)

**Compute modulus division by a power-of-2-number**

Compute n modulo d without division(/) and modulo(%) operators, where d is a power of 2 number.

**Input:** 6 4  
**Output:** 2   
**Explanation:** As 6%4 = 2

**Input:** 12 8  
**Output:** 4  
**Explanation:** As 12%8 = 4

**Input:** 10 2  
**Output:** 0  
**Explanation:**As 10%2 = 0

Let *i*th bit from right is set in d. For getting n modulus d, we just need to return 0 to*i*-1 (from right) bits of n as they are and other bits as 0.

For example if n = 6 (00..110) and d = 4(00..100). Last set bit in d is at position 3 (from right side). So we need to return last two bits of n as they are and other bits as 0, i.e., 00..010.

Now doing it is so easy, guess it….

Yes, you have guessing it right. See the below program.

# Python code to demonstrate

# modulus division by power of 2

# This function will

# return n % d.

# d must be one of:

# 1, 2, 4, 8, 16, 32, …

**def** getModulo(n, d):

**return** ( n & (d**-**1) )

# Driver program to

# test above function

n **=** 6

#d must be a power of 2

d **=** 4

**print**(n,"modulo",d,"is",

      getModulo(n, d))

# This code is contributed by

# Smitha Dinesh Semwal

**Output**

6 modulo 4 is 2

**Time Complexity:** O(1), As we are doing single operation which takes constant time.

**Auxiliary Space:** O(1), As constant extra space is used.

**Find the Number Occurring Odd Number of Times**

Given an array of positive integers. All numbers occur an even number of times except one number which occurs an odd number of times. Find the number in O(n) time & constant space.

**Examples :**

***Input :****arr = {1, 2, 3, 2, 3, 1, 3}*

***Output :****3*

***Input :****arr = {5, 7, 2, 7, 5, 2, 5}*

***Output :****5*

Recommended Practice

[Party of Couples](https://practice.geeksforgeeks.org/problems/alone-in-couple5507/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/alone-in-couple5507/1/)

A **Simple Solution** is to run two nested loops. The outer loop picks all elements one by one and the inner loop counts the number of occurrences of the element picked by the outer loop. The time complexity of this solution is O(n2).

Below is the implementation of the brute force approach :

# Python program to find the element occurring

# odd number of times

# function to find the element occurring odd

# number of times

**def** getOddOccurrence(arr, arr\_size):

**for** i **in** range(0,arr\_size):

        count **=** 0

**for** j **in** range(0, arr\_size):

**if** arr[i] **==** arr[j]:

                count**+=**1

**if** (count **%** 2 !**=** 0):

**return** arr[i]

**return -**1

# driver code

arr **=** [2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2 ]

n **=** len(arr)

print(getOddOccurrence(arr, n))

# This code has been contributed by

# Smitha Dinesh Semwal

**Output :**

5

***Time Complexity:****O(n^2)*

***Auxiliary Space:****O(1)*

A **Better Solution** is to use Hashing. Use array elements as a key and their counts as values. Create an empty hash table. One by one traverse the given array elements and store counts. The time complexity of this solution is O(n). But it requires extra space for hashing.

**Program :**

# Python3 program to find the element

# occurring odd number of times

# function to find the element

# occurring odd number of times

**def** getOddOccurrence(arr,size):

    # Defining HashMap in C++

    Hash**=**dict()

    # Putting all elements into the HashMap

**for** i **in** range(size):

        Hash[arr[i]]**=**Hash.get(arr[i],0) **+** 1;

    # Iterate through HashMap to check an element

    # occurring odd number of times and return it

**for** i **in** Hash:

**if**(Hash[i]**%** 2 !**=** 0):

**return** i

**return -**1

# Driver code

arr**=**[2, 3, 5, 4, 5, 2, 4,3, 5, 2, 4, 4, 2]

n **=** len(arr)

# Function calling

**print**(getOddOccurrence(arr, n))

# This code is contributed by mohit kumar

**Output :**

5

**Time Complexity:** O(n)

**Auxiliary Space:**O(n)

The **Best Solution** is to do bitwise XOR of all the elements. XOR of all elements gives us odd occurring elements.

Here **^** is the **XOR** operators;  
**Note :**  
x^0 = x  
x^y=y^x (**Commutative property holds**)  
(x^y)^z = x^(y^z) (**Distributive property holds**)  
x^x=0

Below is the implementation of the above approach.

# Python program to find the element occurring odd number of times

**def** getOddOccurrence(arr):

    # Initialize result

    res **=** 0

    # Traverse the array

**for** element **in** arr:

        # XOR with the result

        res **=** res ^ element

**return** res

# Test array

arr **=** [ 2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2]

print("%d" **%** getOddOccurrence(arr))

**Output :**

5

**Time Complexity:**O(n)

**Auxiliary Space:**O(1)

**Method 3:Using Built-in Python functions:**

* Count the frequencies of every element using the [**Counter**](https://www.geeksforgeeks.org/python-counter-objects-elements/) function
* Traverse in frequency dictionary
* Check which element has an odd frequency.
* Print that element and break the loop

Below is the implementation:

# importing counter from collections

**from** collections **import** Counter

# Python3 implementation to find

# odd frequency element

**def** oddElement(arr, n):

    # Calculating frequencies using Counter

    count\_map **=** Counter(arr)

**for** i **in** range(0, n):

        # If count of element is odd we return

**if** (count\_map[arr[i]] **%** 2 !**=** 0):

**return** arr[i]

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    arr **=** [1, 1, 3, 3, 5, 6, 6]

    n **=** len(arr)

    print(oddElement(arr, n))

# This code is contributed by vikkycirus

**Output:**

5

**Time Complexity:**O(N)

**Auxiliary Space:**O(N)

**Method 4:Using HashSet**

This problem can also be solved using HashSet by traversing the array and inserting element if not already present else deleting the element from the HashSet. So, after the traversal is complete the only element left in the HashSet is the element which is present three times.

# Python3 program to find the element

# occurring odd number of times

# Function to find the element

# occurring odd number of times

**def** getOddOccurrence(arr, N):

     s **=** set()

**for** i **in** range(N):

**if** arr[i] **in** s:

             s.remove(arr[i]);

**else**:

             s.add(arr[i]);

**return** s

# driver code

arr **=** [ 2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2 ];

n **=** len(arr);

# Function calling

print(**\***getOddOccurrence(arr, n));

# This code is contributed by phasing17.

**Output**

5

**Time Complexity:**O(N)

**Auxiliary Space:**O(N)

**Method 5 (Using binary search) :**First sort the array for [binary search function](https://www.geeksforgeeks.org/binary-search-functions-in-c-stl-binary_search-lower_bound-and-upper_bound/) . Then we can find frequency of all array elements using lower\_bound and upper bound . If frequency is odd then print that element .

Below is the implementation of above approach:

# Python implementation of the above approach

**from** bisect **import** bisect\_left, bisect\_right

# Function to find element that occurs

# odd times in the array

**def** getOddOccurrence(arr, n):

    ans **= -**1

    arr.sort()  # sort array for binary search

    i **=** 0

**while** i < n:

        # index of first and last occ of arr[i]

        first\_index **=** bisect\_left(arr, arr[i])

        last\_index **=** bisect\_right(arr, arr[i])

        i **=** last\_index  # assign i to last\_index to avoid printing

        # same element multiple times

        fre **=** last\_index **-** first\_index **+** 1  # finding frequency

        # (occurrences of arr[i] in the array)

**if** fre **%** 2 !**=** 0:

            # if element occurs odd times then add that elements to our answer

            ans **=** arr[i]

**return** ans

# Driver code

arr **=** [2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2]

n **=** len(arr)

# Function call

**print**(getOddOccurrence(arr, n))

# This code is contributed by phasing17

**Output**

5

**Time Complexity:** O(N\*Log2N)

**Auxiliary Space:** O(1)

**Program to find whether a given number is power of 2**

Given a positive integer **n**, write a function to find if it is a power of 2 or not

**Examples:**

***Input :****n = 4*

***Output :****Yes*

***Explanation:****22 = 4*

***Input :****n = 32*

***Output :****Yes*

***Explanation:****25 = 32*

Recommended Problem

Power of 2

To solve the problem follow the below idea:

*A simple method for this is to simply take the log of the number on base 2 and if you get an integer then the number is the power of 2*

Below is the implementation of the above approach:

# Python3 Program to find

# whether a no is

# power of two

**import** math

# Function to check

# Log base 2

**def** Log2(x):

**if** x **==** 0:

**return** false

**return** (math.log10(x) **/**

            math.log10(2))

# Function to check

# if x is power of 2

**def** isPowerOfTwo(n):

**return** (math.ceil(Log2(n)) **==**

            math.floor(Log2(n)))

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Function call

**if**(isPowerOfTwo(31)):

**print**("Yes")

**else**:

**print**("No")

**if**(isPowerOfTwo(64)):

        print("Yes")

**else**:

        print("No")

# This code is contributed

# by mits

**Output**

No  
Yes

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

**Find whether a given number is a power of 2 using the division operator:**

To solve the problem follow the below idea:

*Another solution is to keep dividing the number by two, i.e, do n = n/2 iteratively. In any iteration, if n%2 becomes non-zero and n is not 1 then n is not a power of 2. If n becomes 1 then it is a power of 2.*

Below is the implementation of the above approach:

# Python program to check if given

# number is power of 2 or not

# Function to check if x is power of 2

**def** isPowerOfTwo(n):

**if** (n **==** 0):

**return** False

**while** (n !**=** 1):

**if** (n **%** 2 !**=** 0):

**return** False

        n **=** n **//** 2

**return** True

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Function call

**if**(isPowerOfTwo(31)):

        print('Yes')

**else**:

        print('No')

**if**(isPowerOfTwo(64)):

        print('Yes')

**else**:

**print**('No')

# This code is contributed by Danish Raza

**Output**

No  
Yes

**Time Complexity:**O(log N)

**Auxiliary Space:**O(1)

**Below is the recursive implementation of the above approach:**

# Python program for above approach

# function which checks whether a

# number is a power of 2

**def** powerof2(n):

    # base cases

    # '1' is the only odd number

    # which is a power of 2(2^0)

**if** n **==** 1:

**return** True

    # all other odd numbers are not powers of 2

**elif** n **%** 2 !**=** 0 **or** n **==** 0:

**return** False

    # recursive function call

**return** powerof2(n**/**2)

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

        # Function call

    print(powerof2(64))  # True

**print**(powerof2(12))  # False

# code contributed by Moukthik a.k.a rowdyninja

**Output**

True  
False

**Time Complexity:**O(log N)

**Auxiliary Space:**O(log N)

**Find whether a given number is a power of 2 by checking the count of set bits:**

To solve the problem follow the below idea:

*All power of two numbers has only a one-bit set. So count the no. of set bits and if you get 1 then the number is a power of 2. Please see*[*Count set bits in an integer*](https://www.geeksforgeeks.org/count-set-bits-in-an-integer/)*for counting set bits.*

Below is the implementation of the above approach:

# Python3 program to check if given

# number is power of 2 or not

# Function to check if x is power of 2

**def** isPowerOfTwo(n):

    cnt **=** 0

**while** n > 0:

**if** n & 1 **==** 1:

            cnt **=** cnt **+** 1

        n **=** n >> 1

**if** cnt **==** 1:

**return** 1

**return** 0

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Function call

**if**(isPowerOfTwo(31)):

        print('Yes')

**else**:

        print('No')

**if**(isPowerOfTwo(64)):

        print('Yes')

**else**:

        print('No')

# This code is contributed by devendra salunke

**Output**

No  
Yes

**Time complexity:**O(log N)

**Auxiliary Space:** O(1)

**Find whether a given number is a power of 2 using the AND(&) operator:**

To solve the problem follow the below idea:

*If we subtract a power of 2 numbers by 1 then all unset bits after the only set bit become set; and the set bit becomes unset.*

*For example for 4 ( 100) and 16(10000), we get the following after subtracting 1*

*3 –> 011*

*15 –> 01111*

*So, if a number n is a power of 2 then bitwise & of n and n-1 will be zero. We can say n is a power of 2 or not based on the value of n&(n-1). The expression n&(n-1) will not work when n is 0. To handle this case also, our expression will become n& (!n&(n-1)) (thanks to*[*https://www.geeksforgeeks.org/program-to-find-whether-a-no-is-power-of-two/*](https://www.geeksforgeeks.org/program-to-find-whether-a-no-is-power-of-two/)*Mohammad for adding this case).*

***Note: With any method which involves subtraction by 1 (just as below, ‘n-1’), some online platforms may give an error if n is given the minimum value of integer or -2147483648. Subtraction by 1 gives integer overflow error in C++.***

Below is the implementation of the above approach:

# Python3 program for the above approach

# Function to check if x is power of 2

**def** isPowerOfTwo(x):

    # First x in the below expression

    # is for the case when x is 0

**return** (x **and** (**not**(x & (x **-** 1))))

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Function call

**if**(isPowerOfTwo(31)):

**print**('Yes')

**else**:

        print('No')

**if**(isPowerOfTwo(64)):

**print**('Yes')

**else**:

        print('No')

# This code is contributed by Danish Raza

**Output**

No  
Yes

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

**Find whether a given number is a power of 2 using the AND(&) and NOT(~) operator:**

To solve the problem follow the below idea:

*Another way is to use the logic to find the rightmost bit set of a given number and then check if (n & (~(n-1))) is equal to n or not*

***Note: With any method which involves subtraction by 1 (just as below, ‘n-1’), some online platforms may give an error if n is given the minimum value of integer or -2147483648. Subtraction by 1 gives integer overflow error in C++.***

Below is the implementation of the above approach:

# Python program of the above approach

# Function to check if x is power of 2\*/

**def** isPowerofTwo(n):

**if** (n **==** 0):

**return** 0

**if** ((n & (~(n **-** 1))) **==** n):

**return** 1

**return** 0

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Function call

**if**(isPowerofTwo(30)):

        print('Yes')

**else**:

**print**('No')

**if**(isPowerofTwo(128)):

        print('Yes')

**else**:

        print('No')

# This code is contributed by shivanisinghss2110

**Output**

No  
Yes

**Time complexity:**O(1)

**Auxiliary Space:**O(1)

**Find whether a given number is a power of 2 using Brian Kernighan’s algorithm:**

To solve the problem follow the below idea:

*As we know that the number which will be the power of two have only one set bit , therefore when we do bitwise AND with the number which is just less than the number which can be represented as the power of (2) then the result will be 0 .*

***Example :****4 can be represented as (2^2 ) ,*

*(4 & 3)=0  or in binary (100 & 011=0)*

***Note: With any method which involves subtraction by 1 (just as below, ‘n-1’), some online platforms may give an error if n is given the minimum value of integer or -2147483648. Subtraction by 1 gives integer overflow error in C++.***

Below is the implementation of the above approach:

# Python3 program of the above approach

# Function to check if x is power of 2

**def** isPowerofTwo(n):

**return** (n !**=** 0) **and** ((n & (n **-** 1)) **==** 0)

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

        # Function call

**if** isPowerofTwo(30):

        print("Yes")

**else**:

        print("No")

**if** isPowerofTwo(128):

        print("Yes")

**else**:

        print("No")

# this code is contributed by aditya942003patil

**Output**

No  
Yes

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

**Find whether a given number is a power of 2 using a floating point bit hack:**

*We can also harness a unique property of*[*IEEE Standard 754*](https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/)*to infer if the given integer is a power of 2 using the following bit hack that only works in a few languages that allow pointer casting.*

*This works because we know that a power of 2 only has 1 bit high and all the other bits low. Therefore if we represent such a number in scientific notation, we’ll always be left with a mantissa of 1. But in IEEE Standard 754 the 1 is discarded from the mantissa as it is redundant. Now we should be left with 0, if not, then the number must not be a power of 2. We will be using double precision.*

***Example:***

*Let’s take 23 first –*

*23 = 00010111*

*=1.0111000 x 2^4*

*Biased Exponent 1023+4=1027*

*1027 = 10000000011*

*Normalised Mantissa = 01110000*

*We will add 0’s to complete the 52 bits*

*The IEEE 754 Double Precision is:*

*= 0 10000000011 0111000000000000000000000000000000000000000000000000*

*Notice that the mantissa is not 0.*

*——————————————————————————————*

*Now let’s take a power of 2, say 16 –*

*16 = 00010000*

*=1.0000000 x 2^4*

*Biased Exponent 1023+4=1027*

*1027 = 10000000011*

*Normalised Mantissa = 00000000*

*We will add 0’s to complete the 52 bits*

*The IEEE 754 Double Precision is:*

*= 0 10000000011 0000000000000000000000000000000000000000000000000000*

*Now the mantissa is strictly 0.*

*——————————————————————————————*

Below is the implementation of the above approach:

**import** math

**def** isPowerOfTwo(x):

    # Power of 2 can't be less than 1.

**if** x < 1:

**return** False

    # check if log2 is an integer using modulo

**return** math.log2(x) **%** 1 **==** 0

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Function calls

    print("Yes" **if** isPowerOfTwo(31) **else** "No")

    print("Yes" **if** isPowerOfTwo(32) **else** "No")

**print**("Yes" **if** isPowerOfTwo(33) **else** "No")

**print**("Yes" **if** isPowerOfTwo(64) **else** "No")

**Output**

No  
Yes  
No  
Yes

**Time Complexity:** O(1)

**Auxiliary Space:** O(1)

**Find position of the only set bit**

Given a number N having only one ‘1’ and all other ’0’s in its binary representation, find position of the only set bit. If there are 0 or more than 1 set bit the answer should be -1. Position of set bit ‘1’ should be counted starting with 1 from the LSB side in the binary representation of the number.

 Source: [Microsoft Interview | 18](https://www.geeksforgeeks.org/microsoft-interview-178/)

**Examples:-**

Input:  
N = 2  
Output:  
2  
Explanation:  
2 is represented as "10" in Binary.  
As we see there's only one set bit  
and it's in Position 2 and thus the  
Output 2.

here is another example

Input:  
N = 5  
Output:  
-1  
Explanation:  
5 is represented as "101" in Binary.  
As we see there's two set bits  
and thus the Output -1.

Find position of set bit

The idea is to start from the rightmost bit and one by one check value of every bit. Following is a detailed algorithm.

**1)**If number is power of two then and then only its binary representation contains only one ‘1’. That’s why check whether the given number is a power of 2 or not. If given number is not a power of 2, then print error message and exit.

**2)** Initialize two variables; i = 1 (for looping) and pos = 1 (to find position of set bit)

**3)** Inside loop, do bitwise AND of i and number ‘N’. If value of this operation is true, then “pos” bit is set, so break the loop and return position. Otherwise, increment “pos” by 1 and left shift i by 1 and repeat the procedure.

# Python3 program to find position of

# only set bit in a given number

# A utility function to check

# whether n is power of 2 or

# not.

**def** isPowerOfTwo(n):

**return** (True **if**(n > 0 **and**

                   ((n & (n **-** 1)) > 0))

**else** False);

# Returns position of the

# only set bit in 'n'

**def** findPosition(n):

**if** (isPowerOfTwo(n) **==** True):

**return -**1;

    i **=** 1;

    pos **=** 1;

    # Iterate through bits of n

    # till we find a set bit i&n

    # will be non-zero only when

    # 'i' and 'n' have a set bit

    # at same position

**while** ((i & n) **==** 0):

        # Unset current bit and

        # set the next bit in 'i'

        i **=** i << 1;

        # increment position

        pos **+=** 1;

**return** pos;

# Driver Code

n **=** 16;

pos **=** findPosition(n);

**if** (pos **== -**1):

    print("n =", n, ", Invalid number");

**else**:

    print("n =", n, ", Position ", pos);

n **=** 12;

pos **=** findPosition(n);

**if** (pos **== -**1):

**print**("n =", n, ", Invalid number");

**else**:

**print**("n =", n, ", Position ", pos);

n **=** 128;

pos **=** findPosition(n);

**if** (pos **== -**1):

    print("n =", n, ", Invalid number");

**else**:

    print("n =", n, ", Position ", pos);

# This code is contributed by mits

**Output :**

n = 16, Position 5  
n = 12, Invalid number  
n = 128, Position 8

**Time Complexity:** O(log2n), where n is the given number

**Space Complexity:**O(1)

Following is **another method** for this problem. The idea is to one by one right shift the set bit of the given number ‘n’ until ‘n’ becomes 0. Count how many times we shifted to make ‘n’ zero. The final count is the position of the set bit.

# Python 3 program to find position

# of only set bit in a given number

# A utility function to check whether

# n is power of 2 or not

**def** isPowerOfTwo(n) :

**return** (n **and** ( **not** (n & (n**-**1))))

# Returns position of the only set bit in 'n'

**def** findPosition(n) :

**if not** isPowerOfTwo(n) :

**return -**1

    count **=** 0

    # One by one move the only set bit to

    # right till it reaches end

**while** (n) :

        n **=** n >> 1

        # increment count of shifts

        count **+=** 1

**return** count

# Driver program to test above function

**if** \_\_name\_\_ **==** "\_\_main\_\_" :

    n **=** 0

    pos **=** findPosition(n)

**if** pos **== -**1 :

**print**("n =", n, "Invalid number")

**else** :

**print**("n =", n, "Position", pos)

    n **=** 12

    pos **=** findPosition(n)

**if** pos **== -**1 :

**print**("n =", n, "Invalid number")

**else** :

        print("n =", n, "Position", pos)

    n **=** 128

    pos **=** findPosition(n)

**if** pos **== -**1 :

**print**("n =", n, "Invalid number")

**else** :

        print("n =", n, "Position", pos)

# This code is contributed by ANKITRAI1

**Output :**

n = 0, Invalid number  
n = 12, Invalid number  
n = 128, Position 8

**Time Complexity:**O(log2n), where n is the given number

**Space Complexity:** O(1)

**We can also use log base 2 to find the position**. Thanks to [Arunkumar](https://www.facebook.com/arunkumar.somalinga)for suggesting this solution.

# Python program to find position

# of only set bit in a given number

**def** Log2n(n):

**if** (n > 1):

**return** (1 **+** Log2n(n **/** 2))

**else**:

**return** 0

# A utility function to check

# whether n is power of 2 or not

**def** isPowerOfTwo(n):

**return** n **and** (**not** (n & (n**-**1)) )

**def** findPosition(n):

**if** (**not** isPowerOfTwo(n)):

**return -**1

**return** Log2n(n) **+** 1

# Driver program to test above function

n **=** 0

pos **=** findPosition(n)

**if**(pos **== -**1):

    print("n =", n, ", Invalid number")

**else**:

**print**("n = ", n, ", Position ", pos)

n **=** 12

pos **=** findPosition(n)

**if**(pos **== -**1):

    print("n =", n, ", Invalid number")

**else**:

**print**("n = ", n, ", Position ", pos)

n **=** 128

pos **=** findPosition(n)

**if**(pos **== -**1):

**print**("n = ", n, ", Invalid number")

**else**:

    print("n = ", n, ", Position ", pos)

# This code is contributed

# by Sumit Sudhakar

**Output :**

n = 0, Invalid number  
n = 12, Invalid number  
n = 128, Position 8

**Time Complexity:**O(log2n)

**Space Complexity:**O(log2n)

**Check for Integer Overflow**

Write a “C” function, int addOvf(int\* result, int a, int b) If there is no overflow, the function places the resultant = sum a+b in “result” and returns 0. Otherwise, it returns -1. The solution of casting to long and adding to find detecting the overflow is not allowed.

**Method 1**

There can be overflow only if signs of two numbers are same, and sign of sum is opposite to the signs of numbers.

1) Calculate sum  
2) If both numbers are positive and sum is negative then return -1  
 Else   
 If both numbers are negative and sum is positive then return -1  
 Else return 0

#include <bits/stdc++.h>

**using namespace** std;

/\* Takes pointer to result and two numbers as

    arguments. If there is no overflow, the function

    places the resultant = sum a+b in “result” and

    returns 0, otherwise it returns -1 \*/

**int** addOvf(**int**\* result, **int** a, **int** b)

{

    \*result = a + b;

**if**(a > 0 && b > 0 && \*result < 0)

**return** -1;

**if**(a < 0 && b < 0 && \*result > 0)

**return** -1;

**return** 0;

}

// Driver code

**int** main()

{

**int** \*res = **new int**[(**sizeof**(**int**))];

**int** x = 2147483640;

**int** y = 10;

    cout<<addOvf(res, x, y);

    cout<<"\n"<<\*res;

**return** 0;

}

// This code is contributed by rathbhupendra

**Output:**

-1  
-2147483646

**Time Complexity:** O(1)

**Space Complexity:** O(1)

**Method 2**

Thanks to Himanshu Aggarwal for adding this method. This method doesn’t modify \*result if there us an overflow.

**import** sys

**def** addOvf(result, a, b):

    # Check if the sum of a and b is greater than INT\_MAX

    # or if a and b are both negative and their sum is less than INT\_MIN

**if** (a >**=** 0 **and** b >**=** 0 **and** (a > sys.maxsize **-** b)) **or** (a < 0 **and** b < 0 **and** (a < **-**sys.maxsize **-** b)):

**return -**1

**else**:

        result[0] **=** a **+** b

**return** 0

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    a **=** sys.maxsize

    b **=** 8192

    res **=** [0]

    print(addOvf(res, a, b))

**Output**

-1

**Time Complexity:** O(1)

**Space Complexity:** O(1)

**Find XOR of two number without using XOR operator**

Given two integers, find XOR of them without using the XOR operator, i.e., without using ^ in C/C++.

**Examples :**

Input: x = 1, y = 2  
Output: 3

Input: x = 3, y = 5  
Output: 6

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

A **Simple Solution** is to traverse all bits one by one. For every pair of bits, check if both are the same, set the corresponding bit like 0 in output, otherwise set it as 1.

# Python3 program to find XOR without using ^

# Returns XOR of x and y

**def** myXOR(x, y):

    res **=** 0 # Initialize result

    # Assuming 32-bit Integer

**for** i **in** range(31, **-**1, **-**1):

        # Find current bits in x and y

        b1 **=** x & (1 << i)

        b2 **=** y & (1 << i)

        b1 **=** min(b1, 1)

        b2 **=** min(b2, 1)

        # If both are 1 then 0

        # else xor is same as OR

        xoredBit **=** 0

**if** (b1 & b2):

            xoredBit **=** 0

**else**:

            xoredBit **=** (b1 | b2)

        # Update result

        res <<**=** 1;

        res |**=** xoredBit

**return** res

# Driver Code

x **=** 3

y **=** 5

print("XOR is", myXOR(x, y))

# This code is contributed by Mohit Kumar

**Output**

XOR is 6

**Time Complexity:** O(**num**), where **num** is the number of bits in the maximum of the two numbers.

**Auxiliary Space:**O(1)

Thanks to Utkarsh Trivedi for suggesting this solution.

A **Better Solution** can find XOR without using a loop.

1) Find bitwise OR of x and y (Result has set bits where either x has set or y has set bit). OR of x = 3 (011) and y = 5 (101) is 7 (111)

2) To remove extra set bits find places where both x and y have set bits. The value of the expression “~x | ~y” has 0 bits wherever x and y both have set bits.

3) bitwise AND of “(x | y)” and “~x | ~y” produces the required result.

Below is the implementation.

# Python 3 program to

# find XOR without using ^

# Returns XOR of x and y

**def** myXOR(x, y):

**return** ((x | y) &

            (~x | ~y))

# Driver Code

x **=** 3

y **=** 5

print("XOR is" ,

       myXOR(x, y))

# This code is contributed

# by Smitha

**Output**

XOR is 6

**Time Complexity:**O(1) i.e. simple calculation of arithmetic and bitwise operator.

**Auxiliary Space:**O(1)

Thanks to **jitu\_the\_best** for suggesting this solution.

**Alternate Solution :**

# Python3 program to

# Returns XOR of x and y

**def** myXOR(x, y):

**return** (x & (~y)) | ((~x )& y)

# Driver Code

x **=** 3

y **=** 5

print("XOR is" ,

    myXOR(x, y))

# This code is contributed by shivanisinghss2110

**Output**

XOR is 6

**Time Complexity:**O(1) i.e. simple calculation of arithmetic and bitwise operator.

**Auxiliary Space:**O(1)

**Another Solution:**we can simply use one of the properties of the XOR bitwise operator i.e. a+b = a^b + 2\*(a&b), with the help of this we can do the same for an operator variant also.

# Python3 program to return XOR of x and y without ^ operator

**def** XOR(x, y):

**return** (x**+**y **-** (2**\***(x & y)))

# Driver Code

x **=** 3

y **=** 5

print("XOR of",x,'&',y,'is:',

      XOR(x, y))

# This code is contributed by vishu05

**Output**

6

**Time Complexity:**O(1) i.e. simple calculation of arithmetic and bitwise operator.

**Auxiliary Space:**O(1)

**Another Solution:**We can simply subtract the **AND**(&) of the two numbers from the **OR**(|) so that the common bit gets canceled and the opposite bits remain in the answer.

# Python program to find XOR without using ^

**def** XOR(x, y):

**return** ((x | y) **-** (x & y))

x, y **=** 3, 5

print(XOR(x, y))

# This code is contributed by lokesh

**Output**

6

**Time Complexity:**O(1) i.e. simple calculation of arithmetic and bitwise operator.

**Auxiliary Space:**O(1)

**Check if two numbers are equal without using arithmetic and comparison operators**

Given two numbers, the task is to check if two numbers are equal without using Arithmetic and Comparison Operators or String functions.

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

**Method 1 :**The idea is to use XOR operator. XOR of two numbers is 0 if the numbers are the same, otherwise non-zero.

# Python3 program to check if two numbers

# are equal without using arithmetic

# and comparison operators

**def** areSame(a, b):

# Function to check if two

# numbers are equal using

# XOR operator

**if** ((a ^ b) !**=** 0):

    print("Not Same")

**else**:

    print("Same")

# Driver Code

areSame(10, 20)

# This code is contributed by Smitha

**Output**

Not Same

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

**Method 2 :**Here idea is using complement ( ~ ) and bit-wise ‘&’ operator.

# Python3 program to check if two numbers

# are equal without using arithmetic

# and comparison operators

# Function to check if two

# numbers are equal using

# using ~ complement and & operator.

**def** areSame(a, b):

**if** ((a & ~b) **==** 0 **and** (~a & b) **==** 0):

        print("Same")

**else**:

        print("Not Same")

# Calling function

areSame(10, 20)

# This code is contributed by Rajput-Ji

**Output**

Not Same

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

**Detect if two integers have opposite signs**

Given two signed integers, write a function that returns true if the signs of given integers are different, otherwise false. For example, the function should return true -1 and +100, and should return false for -100 and -200. The function should not use any of the arithmetic operators.

Let the given integers be x and y. The sign bit is 1 in negative numbers, and 0 in positive numbers. The XOR of x and y will have the sign bit as 1 if they have opposite sign. In other words, XOR of x and y will be negative number if x and y have opposite signs. The following code use this logic.

# Python3 Program to Detect

# if two integers have

# opposite signs.

**def** oppositeSigns(x, y):

**return** ((x ^ y) < 0);

x **=** 100

y **=** 1

**if** (oppositeSigns(x, y) **==** True):

**print** ("Signs are opposite")

**else**:

**print** ("Signs are not opposite")

# This article is contributed by Prerna Saini.

**Output**

Signs are opposite

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

Source: [Detect if two integers have opposite signs](http://graphics.stanford.edu/~seander/bithacks.html#DetectOppositeSigns)

We can also solve this by using two comparison operators. See the following code.

**def** oppositeSigns(x, y):

**return** (y >**=** 0) **if** (x < 0) **else** (y < 0);

# This code is contributed by shivanisingjss2110

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

The first method is more efficient. The first method uses a bitwise XOR and a comparison operator. The second method uses two comparison operators and a bitwise XOR operation is more efficient compared to a comparison operation.

We can also use following method. It doesn’t use any comparison operator. The method is suggested by Hongliang and improved by gaurav.

**def** oppositeSigns(x, y):

**return** ((x ^ y) >> 31)

# this code is contributed by shivanisinghss2110

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

The function is written only for compilers where size of an integer is 32 bit. The expression basically checks sign of (x^y) using bitwise operator ‘>>’. As mentioned above, the sign bit for negative numbers is always 1. The sign bit is the leftmost bit in binary representation. So we need to checks whether the 32th bit (or leftmost bit) of x^y is 1 or not. We do it by right shifting the value of x^y by 31, so that the sign bit becomes the least significant bit. If sign bit is 1, then the value of (x^y)>>31 will be 1, otherwise 0.

# Python Program to detect

# if two integers have opposite signs.

**def** oppositeSigns(x,y):

    product **=** x**\***y

**return** (product<0)

# driver code

x **=** 100

y **= -**100

**if**(oppositeSigns(x, y) **==** True):

  print("Signs are opposite")

**else** :

  print("Signs are not opposite")

# this code is contributed by shinjanpatra

**Output**

Signs are opposite

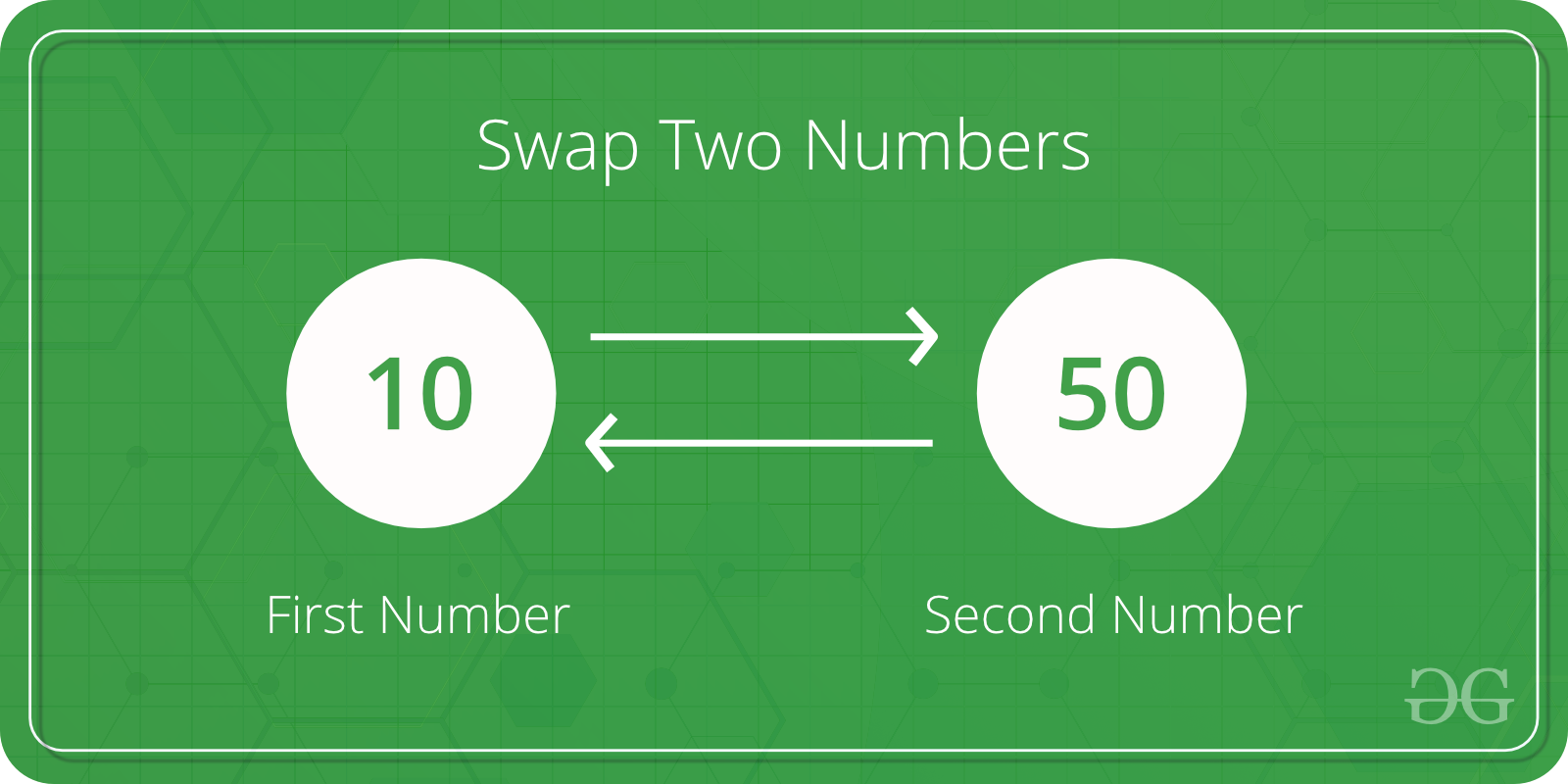
**Approach:** The basic approach is to calculate the product of the two integers, and as we know, two integers having opposite signs will always produce a negative integer, we need to just find out whether the product is negative or not.

**Time Complexity:**O(1)

**Auxiliary Space:**O(1)

**How to swap two numbers without using a temporary variable?**

Given two variables, x, and y, swap two variables without using a third variable.



**Method 1 (Using Arithmetic Operators)**

The idea is to get a sum in one of the two given numbers. The numbers can then be swapped using the sum and subtraction from the sum.

x **=** 10

y **=** 5

# Code to swap 'x' and 'y'

# x now becomes 15

x **=** x **+** y

# y becomes 10

y **=** x **-** y

# x becomes 5

x **=** x **-** y

print("After Swapping: x =", x, " y =", y)

# This code is contributed

# by Sumit Sudhakar

**Output**

After Swapping: x =5, y=10

**Time Complexity:** O(1).

**Auxiliary Space:**O(1).

Multiplication and division can also be used for swapping.

# Python3 program to

# swap two numbers

# without using

# temporary variable

x **=** 10

y **=** 5

# code to swap

# 'x' and 'y'

# x now becomes 50

x **=** x **\*** y

# y becomes 10

y **=** x **//** y;

# x becomes 5

x **=** x **//** y;

**print**("After Swapping: x =",

              x, " y =", y);

# This code is contributed

# by @ajit

**Output**

After Swapping: x =5, y=10

**Time Complexity:** O(1).

**Auxiliary Space:** O(1).

**Method 2 (Using Bitwise XOR)**

The bitwise XOR operator can be used to swap two variables. The XOR of two numbers x and y returns a number that has all the bits as 1 wherever bits of x and y differ. For example, XOR of 10 (In Binary 1010) and 5 (In Binary 0101) is 1111, and XOR of 7 (0111) and 5 (0101) is (0010).

# Python3 code to swap using XOR

x **=** 10

y **=** 5

# Code to swap 'x' and 'y'

x **=** x ^ y; # x now becomes 15 (1111)

y **=** x ^ y; # y becomes 10 (1010)

x **=** x ^ y; # x becomes 5 (0101)

**print** ("After Swapping: x = ", x, " y =", y)

# This code is contributed by

# Sumit Sudhakar

**Output**

After Swapping: x =5, y=10

**Time Complexity:** O(1).

**Auxiliary Space:**O(1).

**Problems with the above methods**

**1)** The multiplication and division-based approach doesn’t work if one of the numbers is 0 as the product becomes 0 irrespective of the other number.

**2)** Both Arithmetic solutions may cause an arithmetic overflow. If x and y are too large, addition and multiplication may go out of the integer range.

**3)** When we use pointers to variable and make a function swap, all the above methods fail when both pointers point to the same variable. Let’s take a look at what will happen in this case if both are pointing to the same variable.

// Bitwise XOR based method

x = x ^ x; // x becomes 0

x = x ^ x; // x remains 0

x = x ^ x; // x remains 0

// Arithmetic based method

x = x + x; // x becomes 2x

x = x – x; // x becomes 0

x = x – x; // x remains 0

Let us see the following program.

**def** swap(xp, yp):

    xp[0] **=** xp[0] ^ yp[0]

    yp[0] **=** xp[0] ^ yp[0]

    xp[0] **=** xp[0] ^ yp[0]

# Driver code

x **=** [10]

swap(x, x)

print("After swap(&x, &x): x = ", x[0])

# This code is contributed by SHUBHAMSINGH10

**Output**

After swap(&x, &x): x = 0

**Time Complexity:** O(1).

**Auxiliary Space:** O(1).

Swapping a variable with itself may be needed in many standard algorithms. For example, see [this](http://geeksquiz.com/quick-sort/)implementation of [QuickSort](http://geeksquiz.com/quick-sort/)where we may swap a variable with itself. The above problem can be avoided by putting a condition before swapping.

# Python3 program of above approach

**def** swap(xp, yp):

    # Check if the two addresses are same

**if** (xp[0] **==** yp[0]):

**return**

    xp[0] **=** xp[0] **+** yp[0]

    yp[0] **=** xp[0] **-** yp[0]

    xp[0] **=** xp[0] **-** yp[0]

# Driver Code

x **=** [10]

swap(x, x)

print("After swap(&x, &x): x = ", x[0])

# This code is contributed by SHUBHAMSINGH10

**Output**

After swap(&x, &x): x = 10

**Time Complexity:**O(1).

**Auxiliary Space:** O(1).

**Method 3 (A mixture of bitwise operators and arithmetic operators)**

The idea is the same as discussed in **Method 1** but uses Bitwise addition and subtraction for swapping.

Below is the implementation of the above approach.

# Python3 program to swap two numbers

# Function to swap the numbers

**def** swap(a, b):

    # Same as a = a + b

    a **=** (a & b) **+** (a | b)

    # Same as b = a - b

    b **=** a **+** (~b) **+** 1

    # Same as a = a - b

    a **=** a **+** (~b) **+** 1

    print("After Swapping: a = ", a, ", b = ", b)

# Driver code

a **=** 5

b **=** 10

# Function call

swap(a, b)

# This code is contributed by bunnyram19

**Output**

After swapping: a = 10, b = 5

**Time Complexity:**O(1)

**Auxiliary Space:**O(1), since no extra space has been taken.

**Method 4 (One Line Expression)**

We can write only one line to swap two numbers.

* x = x ^ y ^ (y = x);
* x = x + y – (y = x);
* x = (x \* y) / (y = x);
* x , y = y, x (In Python)
* C++
* C
* Java
* Python3
* C#
* Javascript

# Python3 program to swap two numbers

# Function to swap the numbers

**def** swap(x, y):

  x , y **=** y, x

**print**("After Swapping: x = ", x, ", y = ", y)

# Driver code

x **=** 10

y **=** 5

# Function call

swap(x, y)

# This code is contributed by kothavvsaakash

**Output**

5 10

**Time Complexity:** O(1)

**Auxiliary Space:** O(1)

**Russian Peasant (Multiply two numbers using bitwise operators)**

Given two integers, write a function to multiply them without using multiplication operator.

There are many other ways to multiply two numbers (For example, see [this](https://www.geeksforgeeks.org/multiply-two-numbers-without-using-multiply-division-bitwise-operators-and-no-loops/)). One interesting method is the [Russian peasant algorithm](http://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication#Russian_peasant_multiplication). The idea is to double the first number and halve the second number repeatedly till the second number doesn’t become 1. In the process, whenever the second number become odd, we add the first number to result (result is initialized as 0)

The following is simple algorithm.

Let the two given numbers be 'a' and 'b'  
1) Initialize result 'res' as 0.  
2) Do following while 'b' is greater than 0  
 a) If 'b' is odd, add 'a' to 'res'  
 b) Double 'a' and halve 'b'  
3) Return 'res'.

# A method to multiply two numbers

# using Russian Peasant method

# Function to multiply two numbers

# using Russian Peasant method

**def** russianPeasant(a, b):

    res **=** 0 # initialize result

    # While second number doesn't

    # become 1

**while** (b > 0):

        # If second number becomes

        # odd, add the first number

        # to result

**if** (b & 1):

            res **=** res **+** a

        # Double the first number

        # and halve the second

        # number

        a **=** a << 1

        b **=** b >> 1

**return** res

# Driver program to test

# above function

**print**(russianPeasant(18, 1))

print(russianPeasant(20, 12))

# This code is contributed by

# Smitha Dinesh Semwal

Output:

18  
240

***Time Complexity:****O(log2b)*

***Auxiliary Space:****O(1)*

**How does this work?**

The value of a\*b is same as (a\*2)\*(b/2) if b is even, otherwise the value is same as ((a\*2)\*(b/2) + a). In the while loop, we keep multiplying ‘a’ with 2 and keep dividing ‘b’ by 2. If ‘b’ becomes odd in loop, we add ‘a’ to ‘res’. When value of ‘b’ becomes 1, the value of ‘res’ + ‘a’, gives us the result.

Note that when ‘b’ is a power of 2, the ‘res’ would remain 0 and ‘a’ would have the multiplication. See the reference for more information.