# Bit Manipulation

## Count set bits in an integer

Given a positive integer N, print count of set bits in it.

**Example 1:**

**Input:**

N = 6

**Output:**

2

**Explanation:**

Binary representation is '110'

So the count of the set bit is 2.

**Example 2:**

**Input:**

8

**Output:**

1

**Explanation:**

Binary representation is '1000'

So the count of the set bit is 1.

**Your Task:**    
You don't need to read input or print anything. Your task is to complete the function **setBits**() which takes an Integer N and returns the count of number of set bits.

**Expected Time Complexity:** O(LogN)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ N ≤ 109

### Solution

**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.

// C++ program to Count set

// bits in an integer

#include <bits/stdc++.h>

using namespace std;

/\* Function to get no of set bits in binary

representation of positive integer n \*/

unsigned int countSetBits(unsigned int n)

{

unsigned int count = 0;

while (n) {

count += n & 1;

n >>= 1;

}

return count;

}

/\* Program to test function countSetBits \*/

int main()

{

int i = 9;

cout << countSetBits(i);

return 0;

}

**Output :**

2

**Time Complexity:** Θ(logn) (Theta of logn)

**Auxiliary Space:**O(1)

**Recursive Approach:**

// cpp implementation of recursive

// approach to find the number

// of set bits in binary representation

// of positive integer n

#include <bits/stdc++.h>

using namespace std;

// recursive function to count set bits

int countSetBits(int n)

{

// base case

if (n == 0)

return 0;

else

// if last bit set add 1 else add 0

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

}

// driver code

int main()

{

// get value from user

int n = 9;

// function calling

cout << countSetBits(n);

return 0;

}

**Output :**

2

**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

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

// C++ program to Count set

// bits in an integer

#include <iostream>

using namespace std;

class gfg {

/\* Function to get no of set bits in binary

representation of passed binary no. \*/

public:

unsigned int countSetBits(int n)

{

unsigned int count = 0;

while (n) {

n &= (n - 1);

count++;

}

return count;

}

};

/\* Program to test function countSetBits \*/

int main()

{

gfg g;

int i = 9;

cout << g.countSetBits(i);

return 0;

}

**Output :**

2

**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.

**Time Complexity:** O(logn)

**Recursive Approach:**

// CPP implementation for recursive

// approach to find the number of set

// bits using Brian Kernighan’s Algorithm

#include <bits/stdc++.h>

using namespace std;

// recursive function to count set bits

int countSetBits(int n)

{

// base case

if (n == 0)

return 0;

else

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

}

// driver code

int main()

{

// get value from user

int n = 9;

// function calling

cout << countSetBits(n);

return 0;

}

**Output :**

2

**3. Using Lookup table:**We can count bits in O(1) time using the lookup table.  
Below is the implementation of the above approach:

// C++ implementation of the approach

#include <bits/stdc++.h>

using namespace std;

int BitsSetTable256[256];

// Function to initialise the lookup table

void initialize()

{

// To initially generate the

// table algorithmically

BitsSetTable256[0] = 0;

for (int i = 0; i < 256; i++)

{

BitsSetTable256[i] = (i & 1) +

BitsSetTable256[i / 2];

}

}

// Function to return the count

// of set bits in n

int countSetBits(int n)

{

return (BitsSetTable256[n & 0xff] +

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

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

BitsSetTable256[n >> 24]);

}

// Driver code

int main()

{

// Initialise the lookup table

initialize();

int n = 9;

cout << countSetBits(n);

}

**Output:**

2

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.

// C++ program to demonstrate \_\_builtin\_popcount()

#include <iostream>

using namespace std;

int main()

{

cout << \_\_builtin\_popcount(4) << endl;

cout << \_\_builtin\_popcount(15);

return 0;

}

**Output :**

1

4

**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.

// C++ program to count set bits by pre-storing

// count set bits in nibbles.

#include <bits/stdc++.h>

using namespace std;

int num\_to\_bits[16] = { 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 \*/

unsigned int countSetBitsRec(unsigned int num)

{

int 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

int main()

{

int num = 31;

cout << countSetBitsRec(num);

return 0;

}

**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.

#include <iostream>

using namespace std;

// Check each bit in a number is set or not

// and return the total count of the set bits.

int countSetBits(int N)

{

int count = 0;

// (1 << i) = pow(2, i)

for (int i = 0; i < sizeof(int) \* 8; i++) {

if (N & (1 << i))

count++;

}

return count;

}

int main()

{

int N = 15;

cout << countSetBits(N) << endl;

return 0;

}

**Output**

4

## Find the two non-repeating elements in an array of repeating elements

Given an array A containing 2\*N+2 positive numbers, out of which 2\*N numbers exist in pairs whereas the other two number occur exactly once and are distinct. Find the other two numbers.

**Example 1:**

**Input:**

N = 2

arr[] = {1, 2, 3, 2, 1, 4}

**Output:**

3 4

**Explanation:**

3 and 4 occur exactly once.

**Example 2:**

**Input:**

N = 1

arr[] = {2, 1, 3, 2}

**Output:**

1 3

**Explanation:**

1 3 occur exactly once.

**Your Task:**  
You do not need to read or print anything. Your task is to complete the function **singleNumber()**which takes the array as input parameter and returns a list of two numbers which occur exactly once in the array. The list must be in ascending order.

**Expected Time Complexity:** O(N)  
**Expected Space Complexity:**O(1)

**Constraints:**  
1 <= length of array <= 106  
1 <= Elements in array <= 5 \* 106

### Solution:

**Method 1(Use Sorting)**   
First, sort all the elements. In the sorted array, by comparing adjacent elements we can easily get the non-repeating elements. Time complexity of this method is O(nLogn)

**Method 2(Use XOR)**   
Let x and y be the non-repeating elements we are looking for and arr[] be the input array. First, calculate the XOR of all the array elements.

xor = arr[0]^arr[1]^arr[2].....arr[n-1]

All the bits that are set in xor will be set in one non-repeating element (x or y) and not in others. So if we take any set bit of xor and divide the elements of the array in two sets – one set of elements with same bit set and another set with same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in the first set, we will get the first non-repeating element, and by doing same in other sets we will get the second non-repeating element.

Let us see an example.

arr[] = {2, 4, 7, 9, 2, 4}

1) Get the XOR of all the elements.

xor = 2^4^7^9^2^4 = 14 (1110)

2) Get a number which has only one set bit of the xor.

Since we can easily get the rightmost set bit, let us use it.

set\_bit\_no = xor & ~(xor-1) = (1110) & ~(1101) = 0010

Now set\_bit\_no will have only set as rightmost set bit of xor.

3) Now divide the elements in two sets and do xor of

elements in each set and we get the non-repeating

elements 7 and 9. Please see the implementation for this step.

**Approach :**  
**Step 1:** Xor all the elements of the array into a variable sum thus all the elements present twice in an array will get removed as for example, 4 = “100” and if 4 xor 4 => “100” xor “100” thus answer will be “000”.   
**Step 2:**Thus in the sum the final answer will be 3 xor 5 as both 2 and 4 are xor with itself giving 0, therefore sum = “011” xor “101” i.e sum = “110” = 6.   
**Step 3:**Now we will take 2’s Complement of sum i.e (-sum) = “010”.   
**Step 4:** Now bitwise And the 2’s of sum with the sum i.e “110” & “010” gives the answer “010” (Aim for bitwise & is that we want to get a number that contains only the rightmost set bit of the sum).   
**Step 5:** bitwise & all the elements of the array with this obtained sum, 2 = “010” & “010” = 2, 3 = “011” & “010” = “010” , 4 = “100” & “010” = “000”, 5 = “101” & “010” = “000”.   
**Step 6:**As we can see that the bitwise & of 2,3 > 0 thus they will be xor with sum1 and bitwise & of 4,5 is resulting into 0 thus they will be xor with sum2.   
**Step 7:**As 2 is present two times so getting xor with sum1 two times only the result 3 is being stored in it and As 4 is also present two times thus getting xor with sum2 will cancel it’s value and thus only 5 will remain there.

**Implementation:**

// C++ program for above approach

#include <bits/stdc++.h>

using namespace std;

/\* This function sets the values of

\*x and \*y to non-repeating elements

in an array arr[] of size n\*/

void get2NonRepeatingNos(int arr[], int n, int\* x, int\* y)

{

/\* Will hold Xor of all elements \*/

int Xor = arr[0];

/\* Will have only single set bit of Xor \*/

int set\_bit\_no;

int i;

\*x = 0;

\*y = 0;

/\* Get the Xor of all elements \*/

for (i = 1; i < n; i++)

Xor ^= arr[i];

/\* Get the rightmost set bit in set\_bit\_no \*/

set\_bit\_no = Xor & ~(Xor - 1);

/\* Now divide elements in two sets by

comparing rightmost set bit of Xor with bit

at same position in each element. \*/

for (i = 0; i < n; i++) {

/\*Xor of first set \*/

if (arr[i] & set\_bit\_no)

\*x = \*x ^ arr[i];

/\*Xor of second set\*/

else {

\*y = \*y ^ arr[i];

}

}

}

/\* Driver code \*/

int main()

{

int arr[] = { 2, 3, 7, 9, 11, 2, 3, 11 };

int n = sizeof(arr) / sizeof(\*arr);

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

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

get2NonRepeatingNos(arr, n, x, y);

cout << "The non-repeating elements are " << \*x

<< " and " << \*y;

}

**Output**

The non-repeating elements are 7 and 9

**Time Complexity:**O(n)   
**Auxiliary Space:** O(1)

**Method 3(Use Maps)**

In this method, we simply count frequency of each element. The elements whose frequency is equal to 1 is the number which is non-repeating. The solution is explained below in the code-

// C++ program for Find the two non-repeating elements in

// an array of repeating elements/ Unique Numbers 2

#include <bits/stdc++.h>

using namespace std;

/\* This function prints the two non-repeating elements in an

\* array of repeating elements\*/

void get2NonRepeatingNos(int arr[], int n)

{

/\*Create map and calculate frequency of array

elements.\*/

map<int, int> m;

for (int i = 0; i < n; i++) {

m[arr[i]]++;

}

/\*Traverse through the map and check if its second

element that is the frequency is 1 or not. If this is

1 than it is the non-repeating element print it.It is

clearly mentioned in problem that all numbers except

two are repeated once. So they will be printed\*/

cout << "The non-repeating elements are ";

for (auto& x : m) {

if (x.second == 1) {

cout << x.first << " ";

}

}

}

/\* Driver code \*/

int main()

{

int arr[] = { 2, 3, 7, 9, 11, 2, 3, 11 };

int n = sizeof(arr) / sizeof(arr[0]);

get2NonRepeatingNos(arr, n);

}

**Output**

The non-repeating elements are 7 9

**Time Complexity**: O(nlogn)   
**Auxiliary Space**: O(n)

**Method 4(Use Sets):**

In this method, We check if the element already exists, if it exists we remove it else we add it to the set.

**Approach**:

**Step 1**: Take each element and check if it exists in the set or not. If it exists go to step-3. If it doesn’t exist go to step-2.

**Step 2**: Add the element to the set and go to step-4.

**Step 3**: Remove the element from the set and go to step-4.

**Step 4**: Print the elements of the set.

**Implementation:**

/\*package whatever //do not write package name here \*/

//Java program to find 2 non repeating elements

//in array that has pairs of numbers

import java.util.LinkedHashSet;

import java.util.Iterator;

import java.io.\*;

class GFG {

//Method to print the 2 non repeating elements in an array

public static void print2SingleNumbers(int[] nums){

// Create a Map Set to store the numbers

LinkedHashSet<Integer> set = new LinkedHashSet<>();

int n = nums.length;

/\*Iterate through the array and check if each

element is present or not in the set. If the

element is present, remove it from the array

otherwise add it to the set\*/

for(int i = 0; i<n; i++){

if(set.contains(nums[i]))

set.remove(nums[i]);

else

set.add(nums[i]);

}

//Iterator is used to traverse through the set

Iterator<Integer> i = set.iterator();

/\*Since there will only be 2 non-repeating elements

we can directly print them\*/

System.out.println("The 2 non repeating numbers are : " + i.next() + " " + i.next());

}

//Driver code

public static void main (String[] args) {

int[] nums = new int[]{2, 3, 7, 9, 11, 2, 3, 11 };

print2SingleNumbers(nums);

}

}

**Output**

The 2 non repeating numbers are : 7 9

**Time Complexity: O(n)**

**Auxiliary Space: O(n)**

## Count number of bits to be flipped to convert A to B

You are given two numbers **A** and **B**. The task is to **count the number of bits needed to be flipped**to **convert**A to B.  
  
**Example 1:**

**Input:** A = 10, B = 20

**Output**: 4

**Explanation**:

A  = 01010

B  = 10100

As we can see, the bits of A that need

to be flipped are **0101**0. If we flip

these bits, we get 10100, which is B.

**Example 2:**

**Input**: A = 20, B = 25

**Output**: 3

**Explanation**:

A  = 10100

B  = 11001

As we can see, the bits of A that need

to be flipped are 1**01**0**0**. If we flip

these bits, we get 11001, which is B.

**Your Task:**The task is to complete the function **countBitsFlip**() that **takes A and B** as parameters and **returns**the **count**of the **number of bits to be flipped** to convert**A to B**.  
  
**Expected Time Complexity:** O(log N).  
**Expected Auxiliary Space:** O(1).  
  
**Constraints:**  
1 ≤ A, B ≤ 106

### Solution:

1. Calculate XOR of A and B.

a\_xor\_b = A ^ B

2. Count the set bits in the above

calculated XOR result.

countSetBits(a\_xor\_b)

XOR of two number will have set bits only at those places where A differs from B.

// Count number of bits to be flipped

// to convert A into B

#include <iostream>

using namespace std;

// Function that count set bits

int countSetBits(int n)

{

int count = 0;

while (n > 0)

{

count++;

n &= (n-1);

}

return count;

}

// Function that return count of

// flipped number

int FlippedCount(int a, int b)

{

// Return count of set bits in

// a XOR b

return countSetBits(a^b);

}

// Driver code

int main()

{

int a = 10;

int b = 20;

cout << FlippedCount(a, b)<<endl;

return 0;

}

**Output**

4

**Time Complexity:** O(logm+logn)

**Space Complexity:** O(1)

**Another approach:**

// C++ program

#include <iostream>

using namespace std;

int countFlips(int a, int b)

{

// initially flips is equal to 0

int flips = 0;

// & each bits of a && b with 1

// and store them if t1 and t2

// if t1 != t2 then we will flip that bit

while(a > 0 || b > 0){

int t1 = (a&1);

int t2 = (b&1);

if(t1!=t2){

flips++;

}

// right shifting a and b

a>>=1;

b>>=1;

}

return flips;

}

int main () {

int a = 10;

int b = 20;

cout <<countFlips(a, b);

}

**Output**

4

**Time Complexity:** O(logm+logn)

**Space Complexity:** O(1)

## Count total set bits in all numbers from 1 to n

You are given a number**N**. Find the **total count of set bits**for all numbers from 1 to N(both inclusive).  
  
**Example 1:**

**Input**: N = 4

**Output**: 5

**Explanation**:

For numbers from 1 to 4.

For 1: 0 0 1 = 1 set bits

For 2: 0 1 0 = 1 set bits

For 3: 0 1 1 = 2 set bits

For 4: 1 0 0 = 1 set bits

Therefore, the total set bits is 5.

**Example 2:**

**Input**: N = 17

**Output**: 35

**Explanation**: From numbers 1 to 17(both inclusive),

the total number of set bits is 35.

**Your Task:**The task is to complete the function **countSetBits**() that takes **n as a parameter**and returns the**count of all bits**.  
  
**Expected Time Complexity:** O(log N).  
**Expected Auxiliary Space:** O(1).  
  
**Constraints:**  
1 ≤ N ≤ 108

### Solution:

**Method 1 (Simple)**   
A simple solution is to run a loop from 1 to n and sum the count of set bits in all numbers from 1 to n.

// A simple program to count set bits

// in all numbers from 1 to n.

#include <iostream>

using namespace std;

// A utility function to count set bits

// in a number x

unsigned int countSetBitsUtil(unsigned int x);

// Returns count of set bits present in all

// numbers from 1 to n

unsigned int countSetBits(unsigned int n)

{

int bitCount = 0; // initialize the result

for (int i = 1; i <= n; i++)

bitCount += countSetBitsUtil(i);

return bitCount;

}

// A utility function to count set bits

// in a number x

unsigned int countSetBitsUtil(unsigned int x)

{

if (x <= 0)

return 0;

return (x % 2 == 0 ? 0 : 1) + countSetBitsUtil(x / 2);

}

// Driver program to test above functions

int main()

{

int n = 4;

cout <<"Total set bit count is " <<countSetBits(n);

return 0;

}

**Output**

Total set bit count is 5

Time Complexity: O(nLogn)

**Method 2 (Simple and efficient than Method 1)**   
If we observe bits from rightmost side at distance i than bits get inverted after 2^i position in vertical sequence.   
for example n = 5;   
0 = 0000   
1 = 0001   
2 = 0010   
3 = 0011   
4 = 0100   
5 = 0101  
Observe the right most bit (i = 0) the bits get flipped after (2^0 = 1)   
Observe the 3rd rightmost bit (i = 2) the bits get flipped after (2^2 = 4)   
So, We can count bits in vertical fashion such that at i’th right most position bits will be get flipped after 2^i iteration;

#include <bits/stdc++.h>

using namespace std;

// Function which counts set bits from 0 to n

int countSetBits(int n)

{

int i = 0;

// ans store sum of set bits from 0 to n

int ans = 0;

// while n greater than equal to 2^i

while ((1 << i) <= n) {

// This k will get flipped after

// 2^i iterations

bool k = 0;

// change is iterator from 2^i to 1

int change = 1 << i;

// This will loop from 0 to n for

// every bit position

for (int j = 0; j <= n; j++) {

ans += k;

if (change == 1) {

k = !k; // When change = 1 flip the bit

change = 1 << i; // again set change to 2^i

}

else {

change--;

}

}

// increment the position

i++;

}

return ans;

}

// Main Function

int main()

{

int n = 17;

cout << countSetBits(n) << endl;

return 0;

}

**Output**

35

Time Complexity: O(k\*n)   
where k = number of bits to represent number n   
k <= 64

**Method 3 (Tricky)**  
If the input number is of the form 2^b -1 e.g., 1, 3, 7, 15.. etc, the number of set bits is b \* 2^(b-1). This is because for all the numbers 0 to (2^b)-1, if you complement and flip the list you end up with the same list (half the bits are on, half off).   
If the number does not have all set bits, then some position m is the position of leftmost set bit. The number of set bits in that position is n – (1 << m) + 1. The remaining set bits are in two parts:  
1) The bits in the (m-1) positions down to the point where the leftmost bit becomes 0, and   
2) The 2^(m-1) numbers below that point, which is the closed form above.  
An easy way to look at it is to consider the number 6:

0|0 0

0|0 1

0|1 0

0|1 1

-|--

1|0 0

1|0 1

1|1 0

The leftmost set bit is in position 2 (positions are considered starting from 0). If we mask that off what remains is 2 (the “1 0” in the right part of the last row.) So the number of bits in the 2nd position (the lower left box) is 3 (that is, 2 + 1). The set bits from 0-3 (the upper right box above) is 2\*2^(2-1) = 4. The box in the lower right is the remaining bits we haven’t yet counted, and is the number of set bits for all the numbers up to 2 (the value of the last entry in the lower right box) which can be figured recursively.

#include <bits/stdc++.h>

// A O(Logn) complexity program to count

// set bits in all numbers from 1 to n

using namespace std;

/\* Returns position of leftmost set bit.

The rightmost position is considered

as 0 \*/

unsigned int getLeftmostBit(int n)

{

int m = 0;

while (n > 1)

{

n = n >> 1;

m++;

}

return m;

}

/\* Given the position of previous leftmost

set bit in n (or an upper bound on

leftmost position) returns the new

position of leftmost set bit in n \*/

unsigned int getNextLeftmostBit(int n, int m)

{

unsigned int temp = 1 << m;

while (n < temp) {

temp = temp >> 1;

m--;

}

return m;

}

// The main recursive function used by countSetBits()

unsigned int \_countSetBits(unsigned int n, int m);

// Returns count of set bits present in

// all numbers from 1 to n

unsigned int countSetBits(unsigned int n)

{

// Get the position of leftmost set

// bit in n. This will be used as an

// upper bound for next set bit function

int m = getLeftmostBit(n);

// Use the position

return \_countSetBits(n, m);

}

unsigned int \_countSetBits(unsigned int n, int m)

{

// Base Case: if n is 0, then set bit

// count is 0

if (n == 0)

return 0;

/\* get position of next leftmost set bit \*/

m = getNextLeftmostBit(n, m);

// If n is of the form 2^x-1, i.e., if n

// is like 1, 3, 7, 15, 31, .. etc,

// then we are done.

// Since positions are considered starting

// from 0, 1 is added to m

if (n == ((unsigned int)1 << (m + 1)) - 1)

return (unsigned int)(m + 1) \* (1 << m);

// update n for next recursive call

n = n - (1 << m);

return (n + 1) + countSetBits(n) + m \* (1 << (m - 1));

}

// Driver code

int main()

{

int n = 17;

cout<<"Total set bit count is "<< countSetBits(n);

return 0;

}

**Output**

Total set bit count is 35

Time Complexity: O(Logn). From the first look at the implementation, time complexity looks more. But if we take a closer look, statements inside while loop of getNextLeftmostBit() are executed for all 0 bits in n. And the number of times recursion is executed is less than or equal to set bits in n. In other words, if the control goes inside while loop of getNextLeftmostBit(), then it skips those many bits in recursion.   
Thanks to agatsu and IC for suggesting this solution.  
Here is another solution suggested by **Piyush Kapoor**.

A simple solution , using the fact that for the ith least significant bit, answer will be

(N/2^i)\*2^(i-1)+ X

where

X = N%(2^i)-(2^(i-1)-1)

iff

N%(2^i)>=(2^(i-1)-1)

int getSetBitsFromOneToN(int N){

int two = 2,ans = 0;

int n = N;

while(n){

ans += (N/two)\*(two>>1);

if((N&(two-1)) > (two>>1)-1) ans += (N&(two-1)) - (two>>1)+1;

two <<= 1;

n >>= 1;

}

return ans;

}

**Method 4 (Recursive)**

Approach:

For each number ‘n’, there will we a number a, a<=n and a is perfect power of two, like 1,2,4,8…..

Let n = 11, now we can see that

Numbers till n, are:

0 -> 0000000

1 -> 0000001

2 -> 0000010

3 -> 0000011

4 -> 0000100

5 -> 0000101

6 -> 0000110

7 -> 0000111

8 -> 0001000

9 -> 0001001

10 -> 0001010

11 -> 0001011

Now we can see that, from 0 to pow(2,1)-1 = 1, we can pair elements top-most with bottom-most,

and count of set bit in a pair is 1

Similarly for pow(2,2)-1 = 4, pairs are:

00 and 11

01 and 10

here count of set bit in a pair is 2, so in both pairs is 4

Similarly we can see for 7, 15, ans soon.....

so we can generalise that,

**count(x) = (x\*pow(2,(x-1)))**,

here x is position of set bit of the largest power of 2 till n

for n = 8, x = 3

for n = 4, x = 2

for n = 5, x = 2

so now for n = 11,

we have added set bits count from 0 to 7 using **count(x) = (x\*pow(2,(x-1)))**

for rest numbers 8 to 11, all will have a set bit at 3rd index, so we can add

count of rest numbers to our ans,

which can be calculated using 11 - 8 + 1 = **(n-pow(2,x) + 1)**

Now if notice that, after removing front bits from rest numbers, we get again number from 0 to some m

so we can recursively call our same function for next set of numbers,

by calling **countSetBits(n - pow(2,x))**

8 -> 1000 -> 000 -> 0

9 -> 1001 -> 001 -> 1

10 -> 1010 -> 010 -> 2

11 -> 1011 -> 011 -> 3

Code:

#include <bits/stdc++.h>

using namespace std;

int findLargestPower(int n)

{

int x = 0;

while ((1 << x) <= n)

x++;

return x - 1;

}

int countSetBits(int n)

{

if (n <= 1)

return n;

int x = findLargestPower(n);

return (x \* pow(2, (x - 1))) + (n - pow(2, x) + 1) + countSetBits(n - pow(2, x));

}

int main()

{

int N = 17;

cout << countSetBits(N) << endl;

return 0;

}

**Output**

35

Time Complexity: O(LogN)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Method 5(Iterative)**

Example:

0  -> 0000000     8  -> 001000      16  -> 010000    24 -> 011000

1  -> 0000001     9  -> 001001      17  -> 010001    25 -> 011001

2  -> 0000010    10  -> 001010     18 -> 010010     26 -> 011010

3  -> 0000011    11  -> 001011     19  -> 010010    27 -> 011011

4  -> 0000100    12  -> 001100     20  -> 010100    28 -> 011100

5  -> 0000101    13  -> 001101     21 -> 010101     29 -> 011101

6  -> 0000110    14  -> 001110     22 -> 010110     30 -> 011110

7  -> 0000111    15  -> 001111     23 -> 010111     31 -> 011111

Input: N = 4

Output: 5

Input: N = 17

Output: 35

Approach : Pattern recognition

Let ‘N’ be any arbitrary number and consider indexing from right to left(rightmost being 1); then  nearestPow = pow(2,i).

Now, when you write all numbers from 1 to N, you will observe the pattern mentioned below:

For every index i, there are exactly nearestPow/2 continuous elements that are unset followed by nearestPow/2 elements that are set.

Throughout the solution, i am going to use this concept.

You can clearly observe above concept in the above table.

The general formula that i came up with:

**a. addRemaining = mod – (nearestPos/2) + 1 iff mod >= nearestPow/2;**

**b. totalSetBitCount = totalRep\*(nearestPow/2) + addRemaining**

   where totalRep -> total number of times the pattern repeats at index i

             addRemaining -> total number of set bits left to be added after the pattern is exhausted

Eg:  let N = 17

       leftMostSetIndex = 5 (Left most set bit index, considering 1 based indexing)

       i = 1 => nearestPos = pow(2,1) = 2;   totalRep = (17+1)/2 = 9 (add 1 only for i=1)

                      mod = 17%2 = 1

                      addRemaining = 0 (only for base case)

                      totalSetBitCount = totalRep\*(nearestPos/2) + addRemaining = 9\*(2/2) + 0 = 9\*1 + 0 = 9

      i = 2 => nearestPos = pow(2, 2)=4;       totalRep = 17/4 = 4

                    mod = 17%4 = 1

                    mod(1) < (4/2) => 1 < 2 => addRemaining = 0

                    totalSetBitCount  = 9 + 4\*(2) + 0 = 9 + 8 = 17

      i = 3 => nearestPow = pow(2,3) = 8;    totalRep = 17/8 = 2

                    mod = 17%8 = 1

                    mod < 4 => addRemaining = 0

                    totalSetBitCount = 17 + 2\*(4) + 0 = 17 + 8 + 0 = 25

    i = 4 => nearestPow = pow(2, 4) = 16; totalRep = 17/16 = 1

                  mod = 17%16 = 1

                   mod < 8 => addRemaining = 0

                 totalSetBitCount  = 25 + 1\*(8) + 0 = 25 + 8 + 0 = 33

We cannot simply operate on the next power(32) as 32>17. Also, as the first half bits will be 0s only, we need to find the distance of the given number(17) from the last power to directly get the number of 1s to be added

   i = 5 => nearestPow = (2, 5) = 32 (base case 2)

                 lastPow = pow(2, 4) = 16

                 mod = 17%16 = 1

                 totalSetBit = 33 + (mod+1) = 33 + 1 + 1 = 35

Therefore, total num of set bits from 1 to 17 is 35

Try iterating with N = 30, for better understanding of the solution.

Solution

#include <iostream>

#include <bits/stdc++.h>

using namespace std;

int GetLeftMostSetBit(int n){

int pos = 0;

while(n>0){

pos++;

n>>=1;

}

return pos;

}

int TotalSetBitsFrom1ToN(int n){

int leftMostSetBitInd = GetLeftMostSetBit(n);

int totalRep, mod;

int nearestPow;

int totalSetBitCount = 0;

int addRemaining=0;

int curr=0; // denotes the number of set bits at index i

//cout<<"leftMostSetBitInd: "<<leftMostSetBitInd<<endl;

for(int i=1; i<=leftMostSetBitInd; ++i){

nearestPow = pow(2, i);

if(nearestPow>n){

int lastPow = pow(2, i-1);

mod = n%lastPow;

totalSetBitCount += mod+1;

}

else{

if(i==1 && n%2==1){

totalRep = (n+1)/nearestPow;

mod = nearestPow%2;

addRemaining = 0;

}

else{

totalRep = n/nearestPow;

mod = n%nearestPow;

if(mod >= (nearestPow/2)){

addRemaining = mod - (nearestPow/2) + 1;

}else{

addRemaining = 0;

}

}

curr = totalRep\*(nearestPow/2) + addRemaining;

totalSetBitCount += curr;

}

// debug output at each iteration

//cout<<i<<" "<<nearestPow<<" "<<totalRep<<" "<<mod<<" "<<totalSetBitCount<<" "<<curr<<endl;

}

return totalSetBitCount;

}

int main(){

std::cout<<TotalSetBitsFrom1ToN(4)<<endl;

std::cout<<TotalSetBitsFrom1ToN(17)<<endl;

std::cout<<TotalSetBitsFrom1ToN(30)<<endl;

return 0;

}

**Output :**

**5**

**35**

**75**

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

**Approach:** Some other approaches to solve this problem has been discussed [here](https://www.geeksforgeeks.org/count-total-set-bits-in-all-numbers-from-1-to-n/). In this article, another approach with time complexity O(logN) has been discussed.   
Check the pattern of Binary representation of the numbers from 1 to N in the following table: 

| Decimal | E | D | C | B | A |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 1 |
| 6 | 0 | 0 | 1 | 1 | 0 |
| 7 | 0 | 0 | 1 | 1 | 1 |
| 8 | 0 | 1 | 0 | 0 | 0 |
| 9 | 0 | 1 | 0 | 0 | 1 |
| 10 | 0 | 1 | 0 | 1 | 0 |
| 11 | 0 | 1 | 0 | 1 | 1 |
| 12 | 0 | 1 | 1 | 0 | 0 |
| 13 | 0 | 1 | 1 | 0 | 1 |
| 14 | 0 | 1 | 1 | 1 | 0 |
| 15 | 0 | 1 | 1 | 1 | 1 |
| 16 | 1 | 0 | 0 | 0 | 0 |

Notice that, 

1. Every alternate bits in A are set.
2. Every 2 alternate bits in B are set.
3. Every 4 alternate bits in C are set.
4. Every 8 alternate bits in D are set.
5. …..
6. This will keep on repeating for every power of 2.

So, we will iterate till the number of bits in the number. And we don’t have to iterate every single number in the range from 1 to n.   
We will perform the following operations to get the desired result. 

* , First of all, we will add 1 to the number in order to compensate 0. As the binary number system starts from 0. So now n = n + 1.
* We will keep the track of the number of set bits encountered till now. And we will initialise it with n/2.
* We will keep one variable which is a power of 2, in order to keep track of bit we are computing.
* We will iterate till the power of 2 becomes greater than n.
* We can get the number of pairs of 0s and 1s in the current bit for all the numbers by dividing n by current power of 2.
* Now we have to add the bits in the set bits count. We can do this by dividing the number of pairs of 0s and 1s by 2 which will give us the number of pairs of 1s only and after that, we will multiply that with the current power of 2 to get the count of ones in the groups.
* Now there may be a chance that we get a number as number of pairs, which is somewhere in the middle of the group i.e. the number of 1s are less than the current power of 2 in that particular group. So, we will find modulus and add that to the count of set bits which will be clear with the help of an example.

**Example:** Consider N = 14   
From the table above, there will be 28 set bits in total from 1 to 14.   
We will be considering 20 as A, 21 as B, 22 as C and 23 as D  
First of all we will add 1 to number N, So now our N = 14 + 1 = 15. 

* Calculation for A (20 = 1)   
  15/2 = 7   
  Number of set bits in A = 7 ————> (i)
* Calculation for B (2^1 = 2)   
  15/2 = 7 => there are 7 groups of 0s and 1s   
  Now, to compute number of groups of set bits only, we have to divide that by 2.   
  So, 7/2 = 3. There are 3 set bit groups.   
  And these groups will contain set bits equal to power of 2 this time, which is 2. So we will multiply number of set bit groups with power of 2   
  => 3\*2 = 6 —>(2i)   
  Plus   
  There may be some extra 1s in this because 4th group is not considered, as this division will give us only integer value. So we have to add that as well. Note: – This will happen only when number of groups of 0s and 1s is odd.   
  15%2 = 1 —>(2ii)   
  2i + 2ii => 6 + 1 = 7 ————>(ii)
* Calculation for C (2^2 = 4)   
  15/4 = 3 => there are 3 groups of 0s and 1s   
  Number of set bit groups = 3/2 = 1   
  Number of set bits in those groups = 1\*4 = 4 —> (3i)   
  As 3 is odd, we have to add bits in the group which is not considered   
  So, 15%4 = 3 —> (3ii)   
  3i + 3ii = 4 + 3 = 7 ————>(iii)
* Calculation for D (2^3 = 8)   
  15/8 = 1 => there is 1 group of 0s and 1s. Now in this case there is only one group and that too of only 0.   
  Number of set bit groups = 1/2 = 0   
  Number of set bits in those groups = 0 \* 8 = 0 —> (4i)   
  As number of groups are odd,   
  So, 15%8 = 7 —> (4ii)   
  4i + 4ii = 0 + 7 = 7 ————>(iv)

At this point, our power of 2 variable becomes greater than the number, which is 15 in our case. (power of 2 = 16 and 16 > 15). So the loop gets terminated here.   
Final output = i + ii + iii + iv = 7 + 7 + 7 + 7 = 28   
Number of set bits from 1 to 14 are 28.  
Below is the implementation of the above approach:

// C++ implementation of the approach

#include <iostream>

using namespace std;

// Function to return the sum of the count

// of set bits in the integers from 1 to n

int countSetBits(int n)

{

// Ignore 0 as all the bits are unset

n++;

// To store the powers of 2

int powerOf2 = 2;

// To store the result, it is initialized

// with n/2 because the count of set

// least significant bits in the integers

// from 1 to n is n/2

int cnt = n / 2;

// Loop for every bit required to represent n

while (powerOf2 <= n) {

// Total count of pairs of 0s and 1s

int totalPairs = n / powerOf2;

// totalPairs/2 gives the complete

// count of the pairs of 1s

// Multiplying it with the current power

// of 2 will give the count of

// 1s in the current bit

cnt += (totalPairs / 2) \* powerOf2;

// If the count of pairs was odd then

// add the remaining 1s which could

// not be groupped together

cnt += (totalPairs & 1) ? (n % powerOf2) : 0;

// Next power of 2

powerOf2 <<= 1;

}

// Return the result

return cnt;

}

// Driver code

int main()

{

int n = 14;

cout << countSetBits(n);

return 0;

}

**Output:**

28

## Program to find whether a no is power of two

Given a non-negative integer **N**. The task is to check if N is a power of **2**. More formally, check if**N**can be expressed as **2x**for some **x.**

**Example 1:**

**Input:** N = 1

**Output:** true

**Explanation:**

1 is equal to 2 raised to 0 (20 = 1).

**Example 2:**

**Input:** N = 98

**Output:** false

**Explanation:**

98 cannot be obtained by any power of 2.

**Your Task:**Your task is to complete the function **isPowerofTwo**() which takes **n**as a parameter and returns **true or false** by **checking** is given number can be represented as a power of two or not.  
  
**Expected Time Complexity:** O(log N).  
**Expected Auxiliary Space:** O(1).  
  
**Constraints:**  
0 ≤ N ≤ 1018

### Solution:

* 1. 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.

// C++ Program to find whether a

// no is power of two

#include<bits/stdc++.h>

using namespace std;

// Function to check if x is power of 2

bool isPowerOfTwo(int n)

{

if(n==0)

return false;

return (ceil(log2(n)) == floor(log2(n)));

}

// Driver program

int main()

{

isPowerOfTwo(31)? cout<<"Yes"<<endl: cout<<"No"<<endl;

isPowerOfTwo(64)? cout<<"Yes"<<endl: cout<<"No"<<endl;

return 0;

}

**Output:**

No

Yes

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

**2.**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.

#include <bits/stdc++.h>

using namespace std;

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

bool isPowerOfTwo(int n)

{

if (n == 0)

return 0;

while (n != 1)

{

if (n%2 != 0)

return 0;

n = n/2;

}

return 1;

}

/\*Driver code\*/

int main()

{

isPowerOfTwo(31)? cout<<"Yes\n": cout<<"No\n";

isPowerOfTwo(64)? cout<<"Yes\n": cout<<"No\n";

return 0;

}

**Output :**

No

Yes

***Time Complexity:****O(log2n)*

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

**3.**Another way is to use this simple recursive solution. It uses the same logic as the above iterative solution but uses recursion instead of iteration.

// C++ program for above approach

#include <bits/stdc++.h>

using namespace std;

// Function which checks whether a

// number is a power of 2

bool powerOf2(int 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

else if (n % 2 != 0 || n ==0)

return false;

// recursive function call

return powerOf2(n / 2);

}

// Driver Code

int main()

{

int n = 64;//True

int m = 12;//False

if (powerOf2(n) == 1)

cout << "True" << endl;

else cout << "False" << endl;

if (powerOf2(m) == 1)

cout << "True" << endl;

else

cout << "False" << endl;

}

**Output**

True

False

***Time Complexity:****O(log2n)*

**4.**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.

**5.**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/>Mohammad for adding this case).

Below is the implementation of this method.

Time complexity : O(1)

Space complexity : O(1)

#include <bits/stdc++.h>

using namespace std;

#define bool int

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

bool isPowerOfTwo (int x)

{

/\* First x in the below expression is for the case when x is 0 \*/

return x && (!(x&(x-1)));

}

/\*Driver code\*/

int main()

{

isPowerOfTwo(31)? cout<<"Yes\n": cout<<"No\n";

isPowerOfTwo(64)? cout<<"Yes\n": cout<<"No\n";

return 0;

}

**Output :**

No

Yes

**6**. Another way is to use the logic to find the rightmost bit set of a given number.

#include <iostream>

using namespace std;

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

bool isPowerofTwo(long long n)

{

if (n == 0)

return 0;

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

return 1;

return 0;

}

/\*Driver code\*/

int main()

{

isPowerofTwo(30) ? cout << "Yes\n" : cout << "No\n";

isPowerofTwo(128) ? cout << "Yes\n" : cout << "No\n";

return 0;

}

**Output**

No

Yes

***Time complexity :****O(1)*

***Space complexity :****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 LSB side in binary representation of the number.

**Example 1:**

**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.

**Example 2:**

**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.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **findPosition()** which takes an integer N as input and returns the answer.

**Expected Time Complexity:** O(log(N))  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
0 <= N <= 108

### Solution:

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.

// C++ program to find position of only set bit in a given number

#include <bits/stdc++.h>

using namespace std;

// A utility function to check whether n is a power of 2 or not.

// See http://goo.gl/17Arj

int isPowerOfTwo(unsigned n)

{

return n && (!(n & (n - 1)));

}

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

int findPosition(unsigned n)

{

if (!isPowerOfTwo(n))

return -1;

unsigned 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)) {

// Unset current bit and set the next bit in 'i'

i = i << 1;

// increment position

++pos;

}

return pos;

}

// Driver program to test above function

int main(void)

{

int n = 16;

int pos = findPosition(n);

(pos == -1) ? cout << "n = " << n << ", Invalid number" << endl : cout << "n = " << n << ", Position " << pos << endl;

n = 12;

pos = findPosition(n);

(pos == -1) ? cout << "n = " << n << ", Invalid number" << endl : cout << "n = " << n << ", Position " << pos << endl;

n = 128;

pos = findPosition(n);

(pos == -1) ? cout << "n = " << n << ", Invalid number" << endl : cout << "n = " << n << ", Position " << pos << endl;

return 0;

}

**Output :**

n = 16, Position 5

n = 12, Invalid number

n = 128, Position 8

**Time Complexity:** O(logn)

**Space Complexity:** O(1)

Following is **another method** for this problem. The idea is to one by one right shift the set bit of 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.

// C++ program to find position of only set bit in a given number

#include <bits/stdc++.h>

using namespace std;

// A utility function to check whether n is power of 2 or not

int isPowerOfTwo(unsigned n)

{

return n && (!(n & (n - 1)));

}

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

int findPosition(unsigned n)

{

if (!isPowerOfTwo(n))

return -1;

unsigned 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;

}

return count;

}

// Driver code

int main(void)

{

int n = 0;

int pos = findPosition(n);

(pos == -1) ? cout<<"n = "<<n<<", Invalid number\n" :

cout<<"n = "<<n<<", Position "<< pos<<endl;

n = 12;

pos = findPosition(n);

(pos == -1) ? cout<<"n = "<<n<<", Invalid number\n" :

cout<<"n = "<<n<<", Position "<< pos<<endl;

n = 128;

pos = findPosition(n);

(pos == -1) ? cout<<"n = "<<n<<", Invalid number\n" :

cout<<"n = "<<n<<", Position "<< pos<<endl;

return 0;

}

**Output :**

n = 0, Invalid number

n = 12, Invalid number

n = 128, Position 8

**We can also use log base 2 to find the position**.

#include <bits/stdc++.h>

using namespace std;

unsigned int Log2n(unsigned int n)

{

return (n > 1) ? 1 + Log2n(n / 2) : 0;

}

int isPowerOfTwo(unsigned n)

{

return n && (!(n & (n - 1)));

}

int findPosition(unsigned n)

{

if (!isPowerOfTwo(n))

return -1;

return Log2n(n) + 1;

}

// Driver code

int main(void)

{

int n = 0;

int pos = findPosition(n);

(pos == -1) ? cout<<"n = "<<n<<", Invalid number\n" :

cout<<"n = "<<n<<", Position "<<pos<<" \n";

n = 12;

pos = findPosition(n);

(pos == -1) ? cout<<"n = "<<n<<", Invalid number\n" :

cout<<"n = "<<n<<", Position "<<pos<<" \n";

n = 128;

pos = findPosition(n);

(pos == -1) ? cout<<"n = "<<n<<", Invalid number\n" :

cout<<"n = "<<n<<", Position "<<pos<<" \n";

return 0;

}

**Output :**

n = 0, Invalid number

n = 12, Invalid number

n = 128, Position 8

**My code:**

int findPosition(int N) {

if(N==0)

return -1;

int res = -1, pos = 1;

while(N>0){

if(N&1==1){

if(res!=-1)

return -1;

res = pos;

}

pos++;

N>>=1;

}

return res;

}

## Copy set bits in a range

Given two numbers x and y, and a range [l, r] where 1 <= l, r <= 32. Find the set bits of y in range [l, r] and set these bits in x also.

**Example 1:**

**Input:**

X = 44, Y = 3

L = 1, R = 5

**Output:** 47

**Explaination:** presenation of 44 and 3 are

101100 and 11. So in the range 1 to 5 there

are two set bits. If those are set in 44

it will become 101111 = 47.

**Example 2:**

**Input:**

X = 16, Y = 2

L = 1, R = 3

**Output:** 18

**Explaination:** representation of 16 and 2

are 10000 and 10. If the mentioned rule is

followed then 16 will become 10010 = 18.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the dunction **setSetBit()** which takes the number x, y, l and r as input parameters and return the modified value of x.

**Expected Time Complexity:** O(r - l)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ x, y ≤ 109  
1 ≤ l ≤ r ≤ 32

### Solution:

**Method 1 (One by one copy bits)**   
We can one by one find set bits of y by traversing given range. For every set bit, we OR it to existing bit of x, so that the becomes set in x, if it was not set. Below is C++ implementation.

// C++ program to rearrange array in alternating

// C++ program to copy set bits in a given

// range [l, r] from y to x.

#include <bits/stdc++.h>

using namespace std;

// Copy set bits in range [l, r] from y to x.

// Note that x is passed by reference and modified

// by this function.

void copySetBits(unsigned &x, unsigned y,

unsigned l, unsigned r)

{

// l and r must be between 1 to 32

// (assuming ints are stored using

// 32 bits)

if (l < 1 || r > 32)

return ;

// Traverse in given range

for (int i=l; i<=r; i++)

{

// Find a mask (A number whose

// only set bit is at i'th position)

int mask = 1 << (i-1);

// If i'th bit is set in y, set i'th

// bit in x also.

if (y & mask)

x = x | mask;

}

}

// Driver code

int main()

{

unsigned x = 10, y = 13, l = 1, r = 32;

copySetBits(x, y, l, r);

cout << "Modified x is " << x;

return 0;

}

**Output**

Modified x is 15

**Time Complexity:** O(r-l)

**Space Complexity:** O(1)

**Method 2 (Copy all bits using one bit mask)**

// C++ program to copy set bits in a given

// range [l, r] from y to x.

#include <bits/stdc++.h>

using namespace std;

// Copy set bits in range [l, r] from y to x.

// Note that x is passed by reference and modified

// by this function.

void copySetBits(unsigned &x, unsigned y,

unsigned l, unsigned r)

{

// l and r must be between 1 to 32

if (l < 1 || r > 32)

return ;

// get the length of the mask

int maskLength = (1ll<<(r-l+1)) - 1;

// Shift the mask to the required position

// "&" with y to get the set bits at between

// l ad r in y

int mask = ((maskLength)<<(l-1)) & y ;

x = x | mask;

}

// Driver code

int main()

{

unsigned x = 10, y = 13, l = 2, r = 3;

copySetBits(x, y, l, r);

cout << "Modified x is " << x;

return 0;

}

**Output**

Modified x is 14

**Time Complexity:** O(1)

**Space Complexity:** O(1)

## Divide two integers without using multiplication, division and mod operator

Given a two integers say a and b. Find the quotient after dividing a by b without using multiplication, division and mod operator.

Input : a = 10, b = 3

Output : 3

Input : a = 43, b = -8

Output : -5

### Solution:

**Approach :**Keep subtracting the dividend from the divisor until dividend becomes less than divisor. The dividend becomes the remainder, and the number of times subtraction is done becomes the quotient. Below is the implementation of above approach :

// C++ implementation to Divide two

// integers without using multiplication,

// division and mod operator

#include <bits/stdc++.h>

using namespace std;

// Function to divide a by b and

// return floor value it

int divide(int dividend, int divisor) {

// Calculate sign of divisor i.e.,

// sign will be negative only iff

// either one of them is negative

// otherwise it will be positive

int sign = ((dividend < 0) ^ (divisor < 0)) ? -1 : 1;

// Update both divisor and

// dividend positive

dividend = abs(dividend);

divisor = abs(divisor);

// Initialize the quotient

int quotient = 0;

while (dividend >= divisor) {

dividend -= divisor;

++quotient;

}

// Return the value of quotient with the appropriate sign.

return quotient \* sign;

}

// Driver code

int main() {

int a = 10, b = 3;

cout << divide(a, b) << "\n";

a = 43, b = -8;

cout << divide(a, b);

return 0;

}

**Output**

3

-5

**Time complexity :**O(a)   
**Auxiliary space :**O(1)

**Efficient Approach:**Use bit manipulation in order to find the quotient. The divisor and dividend can be written as

*dividend = quotient \* divisor + remainder*

As every number can be represented in base 2(0 or 1), represent the quotient in binary form by using shift operator as given below :

1. Determine the most significant bit in the quotient. This can easily be calculated by iterating on the bit position *i* from 31 to 1.
2. Find the first bit for which is less than dividend and keep updating the ith bit position for which it is true.
3. Add the result in *temp* variable for checking the next position such that **(temp + (divisor << i) )** is less than **dividend**.
4. Return the final answer of quotient after updating with corresponding sign.

Below is the implementation of above approach :

// C++ implementation to Divide two

// integers without using multiplication,

// division and mod operator

#include <bits/stdc++.h>

using namespace std;

// Function to divide a by b and

// return floor value it

int divide(long long dividend, long long divisor) {

// Calculate sign of divisor i.e.,

// sign will be negative only iff

// either one of them is negative

// otherwise it will be positive

int sign = ((dividend < 0) ^

(divisor < 0)) ? -1 : 1;

// remove sign of operands

dividend = abs(dividend);

divisor = abs(divisor);

// Initialize the quotient

long long quotient = 0, temp = 0;

// test down from the highest bit and

// accumulate the tentative value for

// valid bit

for (int i = 31; i >= 0; --i) {

if (temp + (divisor << i) <= dividend) {

temp += divisor << i;

quotient |= 1LL << i;

}

}

//if the sign value computed earlier is -1 then negate the value of quotient

if(sign==-1) quotient=-quotient;

return quotient;

}

// Driver code

int main() {

int a = 10, b = 3;

cout << divide(a, b) << "\n";

a = 43, b = -8;

cout << divide(a, b);

return 0;

}

**Output**

3

-5

**Time complexity :**O(log(a))   
**Auxiliary space :**O(1)

## Calculate square of a number without using \*, / and pow()

Given an integer n, calculate the square of a number without using \*, / and pow().

**Examples :**

Input: n = 5

Output: 25

Input: 7

Output: 49

Input: n = 12

Output: 144

### Solution:

A **Simple Solution** is to repeatedly add n to result.

Below is the implementation of this idea.

// Simple solution to calculate square without

// using \* and pow()

#include <iostream>

using namespace std;

int square(int n)

{

// handle negative input

if (n < 0)

n = -n;

// Initialize result

int res = n;

// Add n to res n-1 times

for (int i = 1; i < n; i++)

res += n;

return res;

}

// Driver code

int main()

{

for (int n = 1; n <= 5; n++)

cout << "n = " << n << ", n^2 = " << square(n)

<< endl;

return 0;

}

**Output**

n = 1, n^2 = 1

n = 2, n^2 = 4

n = 3, n^2 = 9

n = 4, n^2 = 16

n = 5, n^2 = 25

The time complexity of the above solution is O(n).

**Approach 2:**

We can do it in **O(Logn) time using bitwise operators**. The idea is based on the following fact.

square(n) = 0 if n == 0

if n is even

square(n) = 4\*square(n/2)

if n is odd

square(n) = 4\*square(floor(n/2)) + 4\*floor(n/2) + 1

Examples

square(6) = 4\*square(3)

square(3) = 4\*(square(1)) + 4\*1 + 1 = 9

square(7) = 4\*square(3) + 4\*3 + 1 = 4\*9 + 4\*3 + 1 = 49

**How does this work?**

If n is even, it can be written as

n = 2\*x

n2 = (2\*x)2 = 4\*x2

If n is odd, it can be written as

n = 2\*x + 1

n2 = (2\*x + 1)2 = 4\*x2 + 4\*x + 1

floor(n/2) can be calculated using a bitwise right shift operator. 2\*x and 4\*x can be calculated

Below is the implementation based on the above idea.

// Square of a number using bitwise operators

#include <bits/stdc++.h>

using namespace std;

int square(int n)

{

// Base case

if (n == 0)

return 0;

// Handle negative number

if (n < 0)

n = -n;

// Get floor(n/2) using right shift

int x = n >> 1;

// If n is odd

if (n & 1)

return ((square(x) << 2) + (x << 2) + 1);

else // If n is even

return (square(x) << 2);

}

// Driver Code

int main()

{

// Function calls

for (int n = 1; n <= 5; n++)

cout << "n = " << n << ", n^2 = " << square(n)

<< endl;

return 0;

}

**Output**

n = 1, n^2 = 1

n = 2, n^2 = 4

n = 3, n^2 = 9

n = 4, n^2 = 16

n = 5, n^2 = 25

The time complexity of the above solution is O(Logn).

**Approach 3:**

For a given number **`num`** we get square of it by multiplying number as **`num \* num`**.

Now write one of **`num`** in square **`num \* num`** in terms of power of **`2`**. Check below examples.

Eg: num = 10, square(num) = 10 \* 10

= 10 \* (8 + 2) = (10 \* 8) + (10 \* 2)

num = 15, square(num) = 15 \* 15

= 15 \* (8 + 4 + 2 + 1) = (15 \* 8) + (15 \* 4) + (15 \* 2) + (15 \* 1)

Multiplication with power of 2's can be done by left shift bitwise operator.

Below is the implementation based on the above idea.

// Simple solution to calculate square without

// using \* and pow()

#include <iostream>

using namespace std;

int square(int num)

{

// handle negative input

if (num < 0) num = -num;

// Initialize result

int result = 0, times = num;

while (times > 0)

{

int possibleShifts = 0, currTimes = 1;

while ((currTimes << 1) <= times)

{

currTimes = currTimes << 1;

++possibleShifts;

}

result = result + (num << possibleShifts);

times = times - currTimes;

}

return result;

}

// Driver code

int main()

{

// Function calls

for (int n = 10; n <= 15; ++n)

cout << "n = " << n << ", n^2 = " << square(n) << endl;

return 0;

}

**Output**

n = 10, n^2 = 100

n = 11, n^2 = 121

n = 12, n^2 = 144

n = 13, n^2 = 169

n = 14, n^2 = 196

n = 15, n^2 = 225

The time complexity of the above solution is O(Log n).

## Power Set

Given a string S find all possible subsequences of the string in lexicographically-sorted order.

**Example 1:**

**Input :** str = "abc"

**Output:** a ab abc ac b bc c

**Explanation :** There are 7 substrings that

can be formed from abc.

**Example 2:**

**Input:** str = "aa"

**Output:** a a aa

**Explanation :** There are 3 substrings that

can be formed from aa.

**Your Task:**  
You don't need to read ot print anything. Your task is to complete the function **AllPossibleStrings()**which takes S as input parameter and returns a list of all possible substrings(non-empty) that can be formed from S in lexicographically-sorted order.

**Expected Time Complexity:**O(2n) where n is the length of the string  
**Expected Space Complexity :**O(n \* 2n)

**Constraints:**  
1 <= Length of string <= 16

### Solution:

vector<string> AllPossibleStrings(string s){

int n = s.size();

vector<string> res;

for(int i=1;i<(1<<n);i++){

string str ="";

for(int j=0;j<n;j++){

if((i&(1<<j))!=0){

str = str + s[j];

}

}

res.push\_back(str);

}

sort(res.begin(),res.end());

return res;

}

**Time Complexity:**O(n2^n)

**Auxiliary Space:**O(1)