**C Programming Bitwise Operators**

This tutorial will explain you the basic idea behind C Programming Bitwise Operators. Why bitwise operators are introduced in C Programming.

## C Programming Bitwise Operators

### **Byte Level Operations**

Till now we have seen all the operators uses variables and constants for expression solving and they operate all the calculations at byte level. like –

res = num1 + num2;

In the above line all are integers which requires two/four/eight bytes memory in C programming depending on the compiler.

### **Bit Level Operations**

1. Sometimes it become mandatory to consider data at bit level.
2. We have to operate on the individual data bit. We also need to turn on/off particular data bit during source code drafting, at that time we must use bitwise operator for doing our task easier.
3. C Programming provides us different bitwise operators for manipulation of bits.
4. Bitwise operators operates on Integer,character but not on float,double
5. Using bitwise operators we can manipulate individual bits easily
6. C programming supports 6 bitwise operators –

### **List of bitwise operators**

|  |  |
| --- | --- |
| **Operator** | **Name of Operator** |
| ~ | One’s Compliment |
| >> | [Right Shift](http://www.c4learn.com/c-programming/c-bitwise-right-shift/) |
| << | [Left Shift](http://www.c4learn.com/c-programming/c-bitwise-left-shift-operator/) |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise XOR |

### **Bitwise operator rules**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input Bits** | | **AND** | **OR** | **XOR** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

## Summary :

| **Operator** | **Usage** |
| --- | --- |
| One's compliment operator | Used to turn a bit on/off |
| AND bitwise operator | Used to mask particular part of byte |
| Left shift bitwise operator | Used to shift the bit to left |
| Right shift bitwise operator | Used to shift the bit to right |

Bitwise Operators : AND,OR,XOR

AND,OR,XOR are three main Bitwise Operators in C Programming Language.AND Operator is used to mask particular Bits.Consider following table –

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input Bits** | | **AND** | **OR** | **XOR** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Example : Consider two numbers 12 and 10

a = 12

b = 10

---------------------------------

a in Binary : 0000 0000 0000 1100

b in Binary : 0000 0000 0000 1010

---------------------------------

a | b : 0000 0000 0000 1110

a & b : 0000 0000 0000 1000

a ^ b : 0000 0000 0000 0110

---------------------------------

**Rules from above table :**

1. If Two bits are same Then Resultant XOR is 0 .
2. If Two bits are different Then Resultant XOR is 1.
3. If any of the bit is 1 then Resultant OR is 1
4. If both bits are 0 then Resultant OR is 0
5. If any of the bit is 0 then Resultant AND is 0.

**Conclusion :**

a | b : 0000 0000 0000 1110 = 14

a & b : 0000 0000 0000 1000 = 8

a ^ b : 0000 0000 0000 0110 = 6

Live Example : Bitwise Operator (AND,OR,XOR)

#include<stdio.h>

**int** main()

{

**int** a=12,b=10;

printf("\nNumber1 AND Number2 : %d",a & b);

printf("\nNumber OR Number2 : %d",a | b);

printf("\nNumber XOR Number2 : %d",a ^ b);

**return**(0);

}

Output :

Number1 AND Number2 : 8

Number OR Number2 : 14

Number XOR Number2 : 6

In the previous chapter we have learnt about [Bitwise Left Shift Operator](http://www.c4learn.com/c-programming/c-bitwise-left-shift-operator/). In this chapter we are looking into Bitwise Right Shift Operator.

## Bitwise Right Shift Operator in C

1. It is denoted by **>>**
2. Bit Pattern of the data can be **shifted by specified number of Positions to Right**
3. When Data is Shifted Right , leading zero’s are **filled with zero**.
4. Right shift Operator is **Binary Operator** [Bi – two]
5. Binary means , **Operator that require two arguments**

### **Quick Overview of Right Shift Operator**

|  |  |
| --- | --- |
| **Original Number A** | 0000  0000  0011  1100 |
| **Right Shift by 2** | **00**00  0000  0000  1111 |
| **Leading 2 Blanks** | Replaced by 0 ,Shown in RED |
| **Direction of Movement of Data** | **Right ========>>>>>>** |

### **Syntax :**

[variable]>>[number of places]

## Live Example : Bitwise Operator [Right Shift Operator]

#include<stdio.h>

**int** main()

{

**int** a = 60;

printf("\nNumber is Shifted By 1 Bit : %d",a >> 1);

printf("\nNumber is Shifted By 2 Bits : %d",a >> 2);

printf("\nNumber is Shifted By 3 Bits : %d",a >> 3);

**return**(0);

}

## Output :

Number is Shifted By 1 Bit : 30

Number is Shifted By 2 Bits : 15

Number is Shifted By 3 Bits : 7

## Bitwise Operations and Masking in C Programming

We have learnt different [Bitwise Operation Techniques](http://www.c4learn.com/c-programming/c-bitwise-operator/), in this chapter we are going to learn the masking technique to set particular bit on or off.  
[box]Masking is the process or operation to set bit on to off or off to on in a byte,nibble or word.[/box]

1. Mask means to block.
2. Masking is the process by which ,only required data is retained and the rest is masked (blocked)
3. Masking can be done using Bitwise Operators
4. Most Commonly Used Bitwise Operator is AND(&)

## A. Masking bits to 1 :

1. In this case we need to retain the particular data.
2. Bitwise OR Operator is used for masking bits to 1

### **Truth table for Bitwise OR :**

| **Bit 1** | **Bit 2** | **Bitwise OR** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |

We can summarize above table as –

| **Bit 1** | **Bit 2** | **Bitwise OR** |
| --- | --- | --- |
| 1 | Y | 1 |
| 0 | Y | Y |

## Live Example : Masking Bits to 1

10011101 10010101

00001000 00001000 OR

-----------------------------

10011101 10011101

## B. Masking bits to 0 :

In this case we need to remove data by masking it to 0

### **Truth table for Bitwise AND :**

| **Bit 1** | **Bit 2** | **Bitwise AND** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |

We can summarize above table as –

| **Bit 1** | **Bit 2** | **Bitwise AND** |
| --- | --- | --- |
| 1 | Y | 1 |
| 0 | Y | Y |

## Masking or Hiding the Last 4 LSB Bits :

Process of Masking : We want Last 4 LSB Bits , So Mask it with (0000 0000 0000 1111)

Num 1 : 1000 0001 1110 1011

& : 0000 0000 0000 1111

--------------------------------

Result : 0000 0000 0000 1011

We get Result as :-

Num 1 : 0000 0000 0000 1011 // First 12 bits are Masked

In this tutorial we will be learning how to bitwise shift negative number using left and right shift. Tutorial explains everything about negative bitwise shifting.

## Bitwise shift negative number

We can Also Shift Negative Number using bitwise operators provided by C (Left Shift and Right Shift). We will learn how to use right shift or left shift operators for Negative Number.

**Recommended article :** [Bitwise left shift](http://www.c4learn.com/c-programming/c-bitwise-left-shift-operator/) | [Bitwise right shift](http://www.c4learn.com/c-programming/c-bitwise-right-shift/)

### **Program #1 : Simple Bitwise shift negative number**

#include<stdio.h>

**void** main()

{

printf("%x",-1<<4);

}

**Output :**

fffffff0

### **Explanation**

In this example we know that –

1. Internal representation of -1 is all 1’s 1111 1111 1111 1111 1111 1111 1111 1111 in an 32 bit compiler.
2. When we bitwise shift negative number by 4 bits to left least significant 4 bits are filled with 0’s
3. Format specifier %x prints specified integer value as hexadecimal format
4. After shifting 1111 1111 1111 1111 1111 1111 1111 0000 = FFFFFFF0 will be printed.

After 1st Shift : [1111 1111 1111 1111 1111 1111 1111 1110]

After 2nd Shift : [1111 1111 1111 1111 1111 1111 1111 1100]

After 3rd Shift : [1111 1111 1111 1111 1111 1111 1111 1000]

After 4th Shift : [1111 1111 1111 1111 1111 1111 1111 0000]

## Some more examples

### **Program #1 : Right bitwise shift negative number**

#include<stdio.h>

**int** main()

{

**int** a = -60;

printf("\nNegative Right Shift by 1 Bit : %d",a >> 1);

printf("\nNegative Right Shift by 2 Bits : %d",a >> 2);

printf("\nNegative Right Shift by 3 Bits : %d",a >> 3);

**return**(0);

}

**Output :**

Negative Right Shift by 1 Bit : -30

Negative Right Shift by 2 Bits : -15

Negative Right Shift by 3 Bits : -8

### **Program #2 : Right bitwise shift negative number**

#include<stdio.h>

**int** main()

{

**int** a = -60;

printf("\nNegative Left Shift by 1 Bit : %d",a << 1);

printf("\nNegative Left Shift by 2 Bits : %d",a << 2);

printf("\nNegative Left Shift by 3 Bits : %d",a << 3);

**return**(0);

}

**Output :**

Negative Left Shift by 1 Bit : -120

Negative Left Shift by 2 Bits : -240

Negative Left Shift by 3 Bits : -480

Bitwise Shifting Long inside Printf :

1. L stands for Long.
2. 1L means “1” is represented in the Long and then we are using Bitwise Shifting of the operators.

Guess the output of the following code ?

main()

{

**int** i;

**for**(i=0;i<5;i++)

printf("%d\n", 1L << i);

}

Output :

1

2

4

8

16

Explanation :

During First Loop

1L << 0

---------------------------------------

0000 0000 0000 0000 0000 0000 0000 0001

During 2nd Loop

1L << 1

---------------------------------------

0000 0000 0000 0000 0000 0000 0000 0010

10 in Decimal = 2

During 3rd Loop

1L << 2

---------------------------------------

0000 0000 0000 0000 0000 0000 0000 0100

100 in Decimal = 4

One’s Compliment Operator in C

1. It is denoted by  ~
2. Bit Pattern of the data can be Reversed using One’s Compliment
3. It inverts each bit of operand .
4. One’s Compliment is Unary Operand i.e Operates on 1 Argument

|  |  |
| --- | --- |
| **Original Number A** | 0000  0000  0011  1100 |
| **One’sCompliment** | 1111  1111  1100  0011 |
| **Zero’s Are Changed to** | 1 |
| **One’s Are Changed to** | 0 |

**Syntax :**

~Variable\_Name

Live Example : Negation Operator in C Programming

#include<stdio.h>

**int** main()

{

**int** a=10;

printf("\nNegation of Number 1 : %d",~a);

**return**(0);

}

Bitwise Left Shift Operator in C

1. It is denoted by **<<**
2. Bit Pattern of the data can be **shifted by specified number of Positions to Left**
3. When Data is Shifted Left , trailing zero’s are **filled with zero**.
4. Left shift Operator is **Binary Operator** [Bi – two]
5. Binary means , **Operator that require two arguments**

Quick Overview of Left Shift Operator

|  |  |
| --- | --- |
| **Original Number A** | 0000  0000  0011  1100 |
| **Left Shift** | 0000  0000  1111  00**00** |
| **Trailing Zero’s** | Replaced by 0 (Shown in RED) |
| **Direction of Movement of Data** | **<<<<<=======Left** |

Syntax : Bitwise Left Shift Operator

[variable]<<[number of places]

Live Example : Bitwise Operator [Left Shift Operator]

#include<stdio.h>

**int** main()

{

**int** a = 60;

printf("\nNumber is Shifted By 1 Bit : %d",a << 1);

printf("\nNumber is Shifted By 2 Bits : %d",a << 2);

printf("\nNumber is Shifted By 3 Bits : %d",a << 3);

**return**(0);

}

Output :

Number is Shifted By 1 Bit : 120

Number is Shifted By 2 Bits : 240

Number is Shifted By 3 Bits : 480

Explanation of Left Shift Binary Operator :

We know the binary representation of the 60 is as below –

0000 0000 0011 1100

Now after shifting all the bits to left towards MSB we will get following bit pattern –

0000 0000 0011 1100 = 60

<< 1

-------------------

0000 0000 0111 1000 = 120

Observations and Conclusion :

|  |  |  |
| --- | --- | --- |
| **Shfting Binary 1 to Left By ‘X’ Bits** | **Decimal Value** | **Converted Value** |
| 0 Bit | 2^0 | 1 |
| 1 Bit | 2^1 | 2 |
| 2 Bits | 2^2 | 4 |
| 3 Bits | 2^3 | 8 |
| 4 Bits | 2^4 | 16 |
| 5 Bits | 2^5 | 32 |
| 6 Bits | 2^6 | 64 |
| 7 Bits | 2^7 | 128 |
| 8 Bits | 2^8 | 256 |

# **BitWise Operators - Basic Tutorial**

[Pankaj Pal](http://edusagar.com/member/profile/1/pankypal)  3 August, 2011  [0](http://edusagar.com/articles/view/8/BitWise-Operators-Basic-Tutorial" \l "disqus_thread) Comments  2.78K

In this article we will discuss about the bitwise operators and their application in practical applications with some examples with required explanation. For simplicity lets stick to C/C++ for this article. The lowest level of data that we can access is 'byte' and by convention a 'byte' contains 8 bits. A 'bit' can represent two different values - 0 and 1. To represent larger values we need collection of bits.

\* ***Bits and HexaDecimal Values***  
   Hexadecimal values contains numbers from 0..9 and then A...F where A=10, B=11 and so on. Thus 0-15 can be represented in hex values from 0-F. To represent values from 0-15, we require 4 bits. Thus to represent a byte in hex values we ultimately require 2 hexadecimal literals. eg.  
  128 in integer == 1000 0000 in binary == 80 in hex OR 0x80 (0x or 0X is used as a prefix to reresent hex values).  
    
\* ***BitWise Operators in C***

  There are six bitwise operators. They are:  
   &   The AND operator  
   |   The OR operator  
   ^   The XOR operator  
   ~   The Ones Complement or Inversion operator  
  >>   The Right Shift operator  
  <<   The Left Shift operator.

    
Hoping that every body is familiar with these operators, hence I am not going to explain the operator and their boolean truth table. Instead lets focus on their application and some good use cases:

\* ***Test whether a particular bit is SET***  
  //check whether 5th bit is set or not in integer value - 50. (count right most bit as the 1st bit.)

  int temp = 50; // 0011\_0010  
  if (temp & 0x10) {  
    // bit 5 is SET   
  }

    
\* ***To Set a Particular 'bit'***  
  //set 3rd bit in an integer with value 50.

  int temp = 50; // 0011\_0010  
  temp = temp | 0x04; //0x04 == 0000\_0100

  Result :: temp -> 0011\_0110

\* ***Toggle bits in an integer***  
  //toggle bit no. 2 from an integer with value - 6.

  int temp = 6; //0000\_0110  
  temp = temp ^ 0x0a   //0000\_1010

    
  This will toggle the 2nd from '1' to '0' and 4th bit from 0 to 1.  
  Result :: temp -> 0000\_1100  
    
\* ***Reset a Particular 'bit'***  
  //reset bit no. 2 from integer with value 50.

  int temp = 50 // 0011\_0110  
  temp = temp & ~0x02 // ~0x02 -> ~(0000\_0010) -> 1111\_1101

    
  Result :: temp -> 0011\_0100  
    
\****multiply by 2n***

  int mult\_by\_pow\_2(int value, int pow)  
  {  
    return value << pow;  
  }

    
  Shifting right a number by 1 effectively multiplies the given number by 2.  
  
  Explanation: 4 << 1 -> 0000\_0100 << 1 -> 0000\_1000 (i.e. '8')  
    
\* ***Finding the bit in 'nth' position is set or not***

  int temp = 50 // 0011\_0110  
  char pos = 5  // bit postion we are interested in.  
  char res;  
    
  res = temp & (1 << pos-1);

    
  Explanation: shifting '1' with 'pos' we are guaranteed to have only a single bit on with all bits as 0, and we know it's to the far-right. Afterwards, the '&' operator will be used to find the particular bit is SET/RESET.

\* ***Check whether a number is power of 2***

  int is\_pow2(int x) {  
    return !(x & (x-1));  
  }

    
  Explanation: A number that is Power of 2 has only a single 'bit' SET. eg. 01000000. Now, doing a (x-1) operation on this values yields all bits set to 1 with the already set bit to 0.  
  
  01000000 - 00000001 = 00111111  
    
  Now, applying an AND operator with the original value  leaves us with all bits reset to 0.  
  01000000 & 00111111 = 00000000.  
    
  Note: 0 as a value is not actually considered as power of 2, hence we can modify the above condition as following to make it complete:

  return (x != 0) && !(x & (x-1));

    
\* ***Calculating the count of SET bits in an integer***

    unsigned int v; // count the number of bits set in v  
    unsigned int c; // c accumulates the total bits set in v  
    for (c = 0; v; c++)  
    {  
      v &= v - 1; // clear the least significant bit set  
    }

      
    Explanation: We have already seen the result of operation (v & (v -1)). This resets the least significant SET bit to 0 in every iteration of while loop. The loop breaks when all the SET bits are reset to 0 and the number of SET bits is the number of iterations of while loop.  
    
Keeping it simple, Lets wind up this article here itself. Most of the examples explained above are fairly simple but they find real use in practical scenarios. For more complex examples related to Bitwise Operator, I have a follow-up article [BitWise Operators Advanced Tricks](http://edusagar.com/articles/view/9/BitWise-Operators-Advanced-Tricks).

# **BitWise Operators - interview Questions**

In this article I am going to present few interview problems that are directly based upon the concepts of bitwise operations. I would urge you to please go back and check [BitWise Operators Basic Tutorial](http://edusagar.com/articles/view/8/BitWise-Operators-Basic-Tutorial) and [BitWise Operators Advanced Tricks](http://edusagar.com/articles/view/9/BitWise-Operators-Advanced-Tricks), as these articles forms the basis of many problems discussed here.  
  
\****Check if a number is a multiple of 3***  
A number is multiple of 3, if the difference between the count of SET bits in odd and even postions are also multiple of 3. Following is a recursive routine in C to accomplish this logic.

int isMult3(unsigned int n)  
{  
    int odd\_c = even\_c = 0; //variables to count odd and even SET bits  
      
    // Terminating condition for the recursive routine.  
    if (n == 0)    // return true if difference is 0.  
    return 1;  
    if (n == 1)    // return false if the difference is not 0.  
    return 0;  
      
    while(n) {  
    if (n&1)   // odd bit is SET, increment odd\_C  
        odd\_c++;  
    n = n >> 1;   
      
    if (n&1)   // even bit is SET, increment even\_c  
        even\_c++;  
    n = n >> 1;  
    }  
      
    // Recursive call this function till you get 0/1 as the difference  
    return(isMult3(abs(odd\_c - even\_c)));  
}

\* ***Multiply by 7 using bitwise operators***

  return ((n << 3) - n );

Explanation: n << 3 will produce the effect of multiplication by 2n i.e. 8. (refer section - [multiply by 2n](http://edusagar.com/articles/view/8/BitWise-Operators-Basic-Tutorial#mult2))  
  
\* ***bit position of rightmost SET bit***  
  
A 2's complement of a number gives us all the bits complemented except the rightmost SET bit. Using this attribute we can arrive at the following code snippet.

    return (log2(n & (-n))) + 1;

Explanation: Anding 2's complement with the original number leaves us with only the righmost bit SET and others as 0. Now, we can take a log base 2 of this binary representation using [Finding the log base 2 of an integer with MSB set](http://edusagar.com/articles/view/9/BitWise-Operators-Advanced-Tricks#log2). Add 1 to get the correct bit position.  
  
\* ***In an array, all the numbers are occurring even number of times, except only one number that occurrs odd number of times.. Find the number.***  
  
a XOR b always return TRUE if and only if a and b are different. We can use this property to figure out the element occurring odd number of times. XORing all the even elements will result in 0 leaving the odd - one out :-).

int getOddNumber(int arr, int arrSize)  
{  
    int i;  
    int temp = 0;  
    for (i = 0; i < arrSize; i++)  
    temp = temp ^ arr[i];  
      
    return temp; //number occurring odd number of times.  
}

\* ***Find out the number of bits that needs to be reversed to transform A to B***  
  
Again, XOR operator is at our disposal, XORing A and B, will SET the bits that are different and hence require bit-reversal in both A and B. After this, we need to count the number of SET bits. (Refer [Calculating the count of SET bits in an integer](http://edusagar.com/articles/view/8/BitWise-Operators-Basic-Tutorial#calcset))

return countSetBits(A ^ B);

\* ***Get A % B where B is power of 2.***  
  
If B is power of 2, then it has only single bit SET. Let us take an example where A=17(0001\_0001) and B=4(0100). To get the result of A % B, we need to return the 2 right-most bits, as these are the remainder if you are going to divide any number with 4(0100).

return (A & (B - 1));

\* ***Circular Rotatation of Bits of a number.***  
  
Circular rotation refers to the shift operation where instead of filling '0' to the bit pattern, we fill the falling bits from the other end.

#define INT\_BITS 32  
/\*left circular rotation of  n by d bits\*/  
int leftCRotate(int n, unsigned int d) {  
    return (n << d)|(n >> (INT\_BITS - d));  
}  
  
/\*Right circular rotation of n by d bits\*/  
int rightCRotate(int n, unsigned int d) {  
    return (n >> d)|(n << (INT\_BITS - d));  
}

That concludes the list of few interview questions based on bitwise operators.

# **BitWise Operators Advanced Tricks**

[Pankaj Pal](http://edusagar.com/member/profile/1/pankypal)  4 August, 2011  [0](http://edusagar.com/articles/view/9/BitWise-Operators-Advanced-Tricks" \l "disqus_thread) Comments  8.22K

This article is an extension of [BitWise Operators - Basic Tutorial](http://edusagar.com/articles/view/8/BitWise-Operators-Basic-Tutorial). In this article we will look at some complex application of bitwise operators.  
  
\* ***Compute the parity of a number***  
  
Parity of a number is based on the count of SET bits ('1') in a number. If the total count of SET bits is even, it is known as even parity, otherwise its known as odd parity.

unsigned int v;       // word value to compute the parity of  
bool parity = false;  // parity will be the parity of v  
  
while (v)  
{  
  parity = !parity;  
  v = v & (v - 1);  
}

Explanation: expression (v & (v-1)) eliminates the left-most SET bit from a number, and the while loop will keep on running till all the SET bits are exhausted. In a way, we are calculating the SET bits in the number. Based on this count we can deduce the parity of the number. It returns 1 if parity is odd, otherwise it returns 0.  
  
  
\* ***Swapping two values without the use of temporary variable***

#define SWAP(a, b) ((&(a) == &(b)) || \  
            (((a) -= (b)), ((b) += (a)), ((a) = (b) - (a))))

Explanation: The check "((&(a) == &(b))" is used here to check whether the memory-locations of the two variables is same or not. Rest all is similar to our normal swap routine, its just that this function uses internal variables from system.  
Note: This should only be used for integers and you might face overflow issues with large values.  
  
There is a slightly faster method to achieve the above operation through the use of XOR operators

#define SWAP(a, b) (((a) ^= (b)), ((b) ^= (a)), ((a) ^= (b)))

\* ***Swapping Individual bits with XOR operator***

unsigned int i, j; // positions of bit sequences to swap  
unsigned int n;    // number of consecutive bits in each sequence  
unsigned int b;    // bits to swap reside in b  
unsigned int r;    // bit-swapped result goes here  
  
unsigned int x = ((b >> i) ^ (b >> j)) & ((1U << n) - 1); // XOR temporary  
r = b ^ ((x << i) | (x << j));

Explanation: Let us take b=0010\_1111, n=3, i=1 (second bit from right) and j = 5. Which means, we have to shift 3(n) consecutive bits from bit-position 1(i) to 5(j). Hence, it results in transition of 0010\_1111 to 11100011.

Step1. First create a mask of as many bits you want to shift which is 3 in this case.  
temp\_a = ((1U << n) - 1) => (1000 - 1) => 0111

Step2. As explained in previous example of swapping using XOR operator, here also we would apply XOR operation on individual 3-bits (1-3 and 5-7) by applying the shift operators. i.e. XOR 111 with 001 (these are the actual bits we want to swap).  
temp\_b = (b >> i) ^ (b >> j) => (0010\_1111 >> 1) ^ (0010\_1111 >> 5) => (0001\_0111) ^ (0000\_0001) => 0001\_0110

Step3. Apply mask created in step1 to get significant 3 bits.  
x = temp\_a & temp\_b => 0000\_0110

Step4. Use the temporary XOR pattern created in Step2 and move it to the desired place. i.e. position 5 to 7 and position 1 to 3.  
temp\_c = (x << i) | (x << j) => (0000\_0110 << 1) | (0000\_0110 << 5) => 0000\_1100 | 1100\_0000 => 1100\_1100

Step5. Carry out the XOR with original number and the XOR pattern to get the desired effect.  
r = b ^ temp\_c => 0010\_1111 ^ 1100\_1100 => 1110\_0011  
  
  
\* ***Reverse the bits***

unsigned int v;     // input bits to be reversed  
unsigned int r = v; // r will be reversed bits of v; first get LSB of v  
int s = sizeof(v) \* CHAR\_BIT - 1; // extra shift needed at end  
  
for (v >>= 1; v; v >>= 1)  
{     
  r <<= 1;  
  r |= v & 1;  
  s--;  
}  
r <<= s; // shift when v's highest bits are zero

Explanation: The bit-reversal routine is pretty straightward. All we are doing is iterating over the whole bit set and then at each step, we are assinging the LSB (right-most bit) into the result. The last step "r <<= s" is neccessary to shift the left-over bits with value '0' which are at the present at the left side of the left-most SET bit (bit with value '1').  
  
Here is a faster version of bit-reversal routine specifically for 32-bit word

unsigned int v; // 32-bit word to reverse bit order  
// swap odd and even bits  
v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1);  
// swap consecutive pairs  
v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2);  
// swap nibbles ...   
v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4);  
// swap bytes  
v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8);  
// swap 2-byte long pairs  
v = ( v >> 16             ) | ( v               << 16);

\* ***Finding the log base 2 of an integer with MSB set***

unsigned int v; // 32-bit input word  
unsigned int r = 0; // result goes here  
while (v >>= 1)  
{  
  r++;  
}

Explanation: The log base 2 of an integer is the same as the position of the highest bit set (or most significant bit set, MSB). e.g. log2(16) => 0001\_0000 => highest SET bit is 4th (if you count LSB as 0) => 4 (the answer).  
  
  
\* ***Finding the log base 2 of an N-bit integer***

unsigned int v;  // 32-bit value to find the log2 of   
const unsigned int b[] = {0x2, 0xC, 0xF0, 0xFF00, 0xFFFF0000};  
const unsigned int S[] = {1, 2, 4, 8, 16};  
int i;  
  
register unsigned int r = 0; // result  
for (i = 4; i >= 0; i--)  
{  
  if (v & b[i])  
  {  
    v >>= S[i];  
    r |= S[i];  
  }   
}

Explanation: The constants defined in the array are actually the expected result of log(num) ie. 2, 4, 8 and so on. The array lists down all the possible values that can represent 2^n for a 32-bit integer. At every step, we are trying to figure out the highest possible value for the operation. If it matches, we store the result in 'r' and then shift it accordingly and wait for the next match if any. If any other match is there, we would OR it with the already computed result 'r'.  
  
  
\* ***Round UP to the next power of 2***

unsigned int v;   
  
v--;  
v |= v >> 1;  //take care of 2-bit word  
v |= v >> 2;  //take care of 4-bit word  
v |= v >> 4;  //take care of 8-bit word  
v |= v >> 8;  //take care of 16-bit word  
v |= v >> 16; //take care of 32-bit word   
v++;

Explanation: Effectively, we have to find the SET bit at the left-most position OR the highest numbered bit which is SET. Now, we need to fill all the bits that are there at the left-hand side of this SET bit with '1'. Next, if we add one to this newly generated number, we are left with a number that is power of 2 and greater then the given number. The expression "v--" is there to handle situation where we pass a number that is already power of 2 to this function.  
  
e.g. Let us take 17 as an example whose binary representation is 0001\_0001.  
To get the next highest power of 2 from this number we have to get to 32 ie. 0010\_0000  
The approach taken in the code above is to mark all the bits in 0001\_0001 from bit-position 4 (counting the left-most bit as 0) to bit-position 0 as '1'.  
We achieve this by the sequential SHIFT and OR operations and finally we would have something like 0001\_1111.  
Lastly, adding 1 to this we get 0010\_0000 which is 32, the next highest power of 2.  
  
Note: The above logic can easily be extended to N-bit words. We will have to provide the shift condition till N/2 bits

# **Count Total Set Bits in All Numbers From 1 to N**

[May 8, 2016 2:31 pm](http://www.crazyforcode.com/count-total-set-bits-numbers-1/) | [Leave a Comment](http://www.crazyforcode.com/count-total-set-bits-numbers-1/#comments) | [crazyadmin](http://www.crazyforcode.com/author/dev1/)

**Problem:**

Given a positive integer n, count the total number of set bits in binary representation of all numbers from 1 to n.

Examples:  
Input: n = 3  
Output: 4

Input: n = 6  
Output: 9

Input: n = 7  
Output: 12

Input: n = 8  
Output: 13

**Solution:**

The solution is to run a loop from 1 to n and sum the count of set bits in all numbers from 1 to n.

|  |  |
| --- | --- |
| int countSetBits(unsigned int n) | |
| { |

|  |  |
| --- | --- |
| unsigned int c; // the total bits set in n | |
| for (c = 0; n; n >>= 1) |

|  |
| --- |
| { |
| c += n & 1; | |

|  |
| --- |
| } |
| return c; | |

|  |
| --- |
| } |

 program on how to toggle alternate bits of an integer using bitwise operators.

#include  
#include  
void main()  
{  
int i, j, k, num, count=1;  
printf(“Enter your number:\r\n”);  
scanf(“%d”, &num);  
if(num==1){  
printf(“Count of bits is 1\r\n”);  
return;  
}  
for(i=1; i>1)&1){  
count++;  
}  
j = j>>1;  
}  
}  
printf(“Count of bits are %d\r\n”,count+1);  
}

# **Find the Maximum of Two Numbers Without Using if-else**

[May 7, 2016 10:57 am](http://www.crazyforcode.com/find-maximum-numbers-if-else/) | [3 Comments](http://www.crazyforcode.com/find-maximum-numbers-if-else/#comments) | [crazyadmin](http://www.crazyforcode.com/author/dev1/)

Find the maximum and minimum of two integers without branching i.e. if condition.

**Solution:**  
Minimum of two numbers can be found from the following:

Min(x,y) = y ^ ((x ^ y) & -(x < y))

It works because if x < y, then -(x < y) will be all ones, so r = y ^ (x ^ y) & ~0 = y ^ x ^ y = x. Otherwise, if x >= y, then -(x < y) will be all zeros, so r = y ^ ((x ^ y) & 0) = y. On some machines, evaluating (x < y) as 0 or 1 requires a branch instruction, so there may be no advantage.

Similarly the maximum of two integers can be found from the following:

Max(x,y) = x ^ ((x ^ y) & -(x < y))

|  |  |
| --- | --- |
| #include<stdio.h> | |
|  |

|  |  |
| --- | --- |
| /\*Function to find minimum of x and y\*/ | |
| int min(int x, int y) |

|  |
| --- |
| { |
| return y ^ ((x ^ y) & -(x < y)); | |

|  |
| --- |
| } |
|  | |

|  |  |
| --- | --- |
| /\*Function to find maximum of x and y\*/ | |
| int max(int x, int y) |

|  |
| --- |
| { |
| return x ^ ((x ^ y) & -(x < y)); | |

|  |
| --- |
| } |

# **Find the Element that Appears Once**

[May 6, 2016 1:07 pm](http://www.crazyforcode.com/find-element-appears/) | [1 Comment](http://www.crazyforcode.com/find-element-appears/#comments) | [crazyadmin](http://www.crazyforcode.com/author/dev1/)

**Problem:** Given an array where every element occurs three times, except one element which occurs only once. Find the element that occurs once. Expected time complexity is O(n) and O(1) extra space.

Examples:  
Input: arr[] = {10, 1, 10, 3, 10, 1, 1, 2, 3, 3}  
Output: 2

**Solution:**

If O(1) space constraint was not there, we could’ve gone for a hashmap with values being the count of occurrences. But since there is space constraint we can go for bitwise operations.

Basically, it makes use of the fact that x^x = 0 and 0^x=x. So all paired elements get XOR’d and vanish leaving the lonely element.  
If a bit is already in ones, add it to twos.  
XOR will add this bit to ones if it’s not there or remove this bit from ones if it’s already there.  
If a bit is in both ones and twos, remove it from ones and twos.  
When finished, ones contains the bits that only appeared 3\*n+1 times, which are the bits for the element that only appeared once.

|  |  |
| --- | --- |
| int getUniqueElement(int[] arr) | |
| { |

|  |  |
| --- | --- |
| //this variable holds XOR of all the elements which have appeared "only" once. | |
| int ones = 0 ; |

|  |  |
| --- | --- |
| //this variable holds XOR of all the elements which have appeared "only" twice. | |
| int twos = 0 ; |

|  |  |
| --- | --- |
| int not\_threes ; | |
|  |

|  |  |
| --- | --- |
| for( int x : arr ) | |
| { |

|  |
| --- |
| twos |= ones & x ; //add it to twos if it exists in ones |
| ones ^= x ; //if it exists in ones, remove, otherwise, add it | |

|  |
| --- |
|  |
| // Next 3 lines of code just converts the common 1's between "ones" and "twos" to zero. | |

|  |
| --- |
|  |
| //if x is in ones and twos, dont add it to Threes. | |

|  |
| --- |
| not\_threes = ~(ones & twos) ; |
| ones &= not\_threes ;//remove x from ones | |

|  |  |
| --- | --- |
| twos &= not\_threes ;//remove x from twos | |
| } |

|  |  |
| --- | --- |
| return ones; | |
| } |

# **Find the Element that Appears Once**

[May 6, 2016 1:07 pm](http://www.crazyforcode.com/find-element-appears/) | [1 Comment](http://www.crazyforcode.com/find-element-appears/#comments) | [crazyadmin](http://www.crazyforcode.com/author/dev1/)

**Problem:** Given an array where every element occurs three times, except one element which occurs only once. Find the element that occurs once. Expected time complexity is O(n) and O(1) extra space.

Examples:  
Input: arr[] = {10, 1, 10, 3, 10, 1, 1, 2, 3, 3}  
Output: 2

**Solution:**

If O(1) space constraint was not there, we could’ve gone for a hashmap with values being the count of occurrences. But since there is space constraint we can go for bitwise operations.

Basically, it makes use of the fact that x^x = 0 and 0^x=x. So all paired elements get XOR’d and vanish leaving the lonely element.  
If a bit is already in ones, add it to twos.  
XOR will add this bit to ones if it’s not there or remove this bit from ones if it’s already there.  
If a bit is in both ones and twos, remove it from ones and twos.  
When finished, ones contains the bits that only appeared 3\*n+1 times, which are the bits for the element that only appeared once.

|  |  |
| --- | --- |
| int getUniqueElement(int[] arr) | |
| { |

|  |  |
| --- | --- |
| //this variable holds XOR of all the elements which have appeared "only" once. | |
| int ones = 0 ; |

|  |  |
| --- | --- |
| //this variable holds XOR of all the elements which have appeared "only" twice. | |
| int twos = 0 ; |

|  |  |
| --- | --- |
| int not\_threes ; | |
|  |

|  |  |
| --- | --- |
| for( int x : arr ) | |
| { |

|  |
| --- |
| twos |= ones & x ; //add it to twos if it exists in ones |
| ones ^= x ; //if it exists in ones, remove, otherwise, add it | |

|  |
| --- |
|  |
| // Next 3 lines of code just converts the common 1's between "ones" and "twos" to zero. | |

|  |
| --- |
|  |
| //if x is in ones and twos, dont add it to Threes. | |

|  |
| --- |
| not\_threes = ~(ones & twos) ; |
| ones &= not\_threes ;//remove x from ones | |

|  |  |
| --- | --- |
| twos &= not\_threes ;//remove x from twos | |
| } |

|  |  |
| --- | --- |
| return ones; | |
| } |

# **Count Number of Bits to be Flipped to Convert A to B**

[February 12, 2016 5:25 pm](http://www.crazyforcode.com/count-number-bits-flipped-convert/) | [1 Comment](http://www.crazyforcode.com/count-number-bits-flipped-convert/#comments) | [crazyadmin](http://www.crazyforcode.com/author/dev1/)

Suppose, I have two numbers A and B. I need to find out how many numbers of bits needed to be changed to convert A to B.

Like:

A = 1101101  
B = 1011011  
^^ ^^  
Here, we need to change 4 bits to convert A to B

How can I do this?

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.

|  |  |
| --- | --- |
| int countSetBits(unsigned int n) { | |
| unsigned int count = 0; |

|  |
| --- |
| while (n) { |
| count += n & 1; | |

|  |  |
| --- | --- |
| n >>= 1; | |
| } |

|  |  |
| --- | --- |
| return count; | |
| } |

|  |
| --- |
|  |
| // get the no. of bits to be flipped to convert 'a' to 'b' | |

|  |
| --- |
| int getNoBits(int a, int b) { |
| // xor of 'a' & 'b' will have set bits at those | |

|  |  |
| --- | --- |
| // places where bits of 'a' and 'b' differs | |
| unsigned int k = a ^ b; |

|  |  |
| --- | --- |
| return countSetBits(k); | |
| } |

# **Find if a no is Power of Two**

[October 22, 2013 10:18 pm](http://www.crazyforcode.com/find-number-power-two/) | [Leave a Comment](http://www.crazyforcode.com/find-number-power-two/#comments) | [crazyadmin](http://www.crazyforcode.com/author/dev1/)

**Problem**: Write a C program to find if a number is of power of 2?

**Solution**:

**Method 1: (Using arithmetic)**

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.

/\* 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; | |
| } |

**Method 2: (Using Bitwise operator)**

If we subtract 1 from a number that is power of 2 then all unset bits after the only set bit become set and the set bit become unset.

For example for 4 ( 100) and 16(10000), we get 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 give zero. We can say n is a power of 2 or not based on value of n &(n-1).

|  |  |
| --- | --- |
| bool IsPowerOfTwo(long x) | |
| { |

|  |  |
| --- | --- |
| return (x & (x - 1)) == 0; | |
| } |

For completeness, zero is not a power of two. If you want to take into account that edge case, here’s how:

|  |  |
| --- | --- |
| bool IsPowerOfTwo(long x) | |
| { |

|  |  |
| --- | --- |
| return (x != 0) && ((x & (x - 1)) == 0); | |
| } |

Lets understand this with example  
bool b = IsPowerOfTwo(4)  
if x = 4

return (4 != 0) && ((4 & (4-1)) == 0);  
Well we already know that 4 != 0 evals to true, so far so good. But what about:

((4 & (4-1)) == 0)  
This translates to this of course:

((4 & 3) == 0)  
But what exactly is 4&3?

The binary representation of 4 is 100 and the binary representation of 3 is 011 (remember the & takes the binary representation of these numbers. So we have:

100 = 4  
011 = 3

1 & 1 = 1, 1 & 0 = 0, 0 & 0 = 0, and 0 & 1 = 0. So we do the math:

100  
011  
—-  
000

# **Find the element that appears once**

Given an array where every element occurs three times, except one element which occurs only once. Find the element that occurs once. Expected time complexity is O(n) and O(1) extra space.  
Examples:

Input: arr[] = {12, 1, 12, 3, 12, 1, 1, 2, 3, 3}

Output: 2

## [Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/alone-in-couple/0)

We can use sorting to do it in O(nLogn) time. We can also use hashing, it has the worst case time complexity of O(n), but requires extra space.

The idea is to use bitwise operators for a solution that is O(n) time and uses O(1) extra space. The solution is not easy like other XOR based solutions, because all elements appear odd number of times here. The idea is taken from [here](http://www.careercup.com/question?id=7902674).

Run a loop for all elements in array. At the end of every iteration, maintain following two values.

ones: The bits that have appeared 1st time or 4th time or 7th time .. etc.

twos: The bits that have appeared 2nd time or 5th time or 8th time .. etc.

Finally, we return the value of ‘ones’

How to maintain the values of ‘ones’ and ‘twos’?  
‘ones’ and ‘twos’ are initialized as 0. For every new element in array, find out the common set bits in the new element and previous value of ‘ones’. These common set bits are actually the bits that should be added to ‘twos’. So do bitwise OR of the common set bits with ‘twos’. ‘twos’ also gets some extra bits that appear third time. These extra bits are removed later.  
Update ‘ones’ by doing XOR of new element with previous value of ‘ones’. There may be some bits which appear 3rd time. These extra bits are also removed later.

Both ‘ones’ and ‘twos’ contain those extra bits which appear 3rd time. Remove these extra bits by finding out common set bits in ‘ones’ and ‘twos’.

* C/C++
* Java

|  |
| --- |
| #include <stdio.h>    int getSingle(int arr[], int n)  {      int ones = 0, twos = 0 ;        int common\_bit\_mask;        // Let us take the example of {3, 3, 2, 3} to understand this      for( int i=0; i< n; i++ )      {          /\* The expression "one & arr[i]" gives the bits that are             there in both 'ones' and new element from arr[].  We             add these bits to 'twos' using bitwise OR               Value of 'twos' will be set as 0, 3, 3 and 1 after 1st,             2nd, 3rd and 4th iterations respectively \*/          twos  = twos | (ones & arr[i]);              /\* XOR the new bits with previous 'ones' to get all bits             appearing odd number of times               Value of 'ones' will be set as 3, 0, 2 and 3 after 1st,             2nd, 3rd and 4th iterations respectively \*/          ones  = ones ^ arr[i];              /\* The common bits are those bits which appear third time             So these bits should not be there in both 'ones' and 'twos'.             common\_bit\_mask contains all these bits as 0, so that the bits can             be removed from 'ones' and 'twos'               Value of 'common\_bit\_mask' will be set as 00, 00, 01 and 10             after 1st, 2nd, 3rd and 4th iterations respectively \*/          common\_bit\_mask = ~(ones & twos);              /\* Remove common bits (the bits that appear third time) from 'ones'               Value of 'ones' will be set as 3, 0, 0 and 2 after 1st,             2nd, 3rd and 4th iterations respectively \*/          ones &= common\_bit\_mask;              /\* Remove common bits (the bits that appear third time) from 'twos'               Value of 'twos' will be set as 0, 3, 1 and 0 after 1st,             2nd, 3rd and 4th itearations respectively \*/          twos &= common\_bit\_mask;            // uncomment this code to see intermediate values          //printf (" %d %d n", ones, twos);      }        return ones;  }    int main()  {      int arr[] = {3, 3, 2, 3};      int n = sizeof(arr) / sizeof(arr[0]);      printf("The element with single occurrence is %d ",              getSingle(arr, n));      return 0;  } |

Run on IDE

Output:

2

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

# **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 iff they have opposite sign. In other words, XOR of x and y will be negative number number iff x and y have opposite signs. The following code use this logic.

|  |
| --- |
| #include<stdbool.h>  #include<stdio.h>    bool oppositeSigns(int x, int y)  {      return ((x ^ y) < 0);  }    int main()  {      int x = 100, y = -100;      if (oppositeSigns(x, y) == true)         printf ("Signs are opposite");      else        printf ("Signs are not opposite");      return 0;  } |

Run on IDE

Output:

Signs are opposite

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.

|  |
| --- |
| bool oppositeSigns(int x, int y)  {      return (x < 0)? (y >= 0): (y < 0);  } |

Run on IDE

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.

|  |
| --- |
| bool oppositeSigns(int x, int y)  {      return ((x ^ y) >> 31);  } |

Run on IDE

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.

# **Add 1 to a given number**

Write a program to add one to a given number. You are not allowed to use operators like ‘+’, ‘-‘, ‘\*’, ‘/’, ‘++’, ‘–‘ …etc.

Examples:  
Input: 12  
Output: 13

Input: 6  
Output: 7

Yes, you guessed it right, we can use bitwise operators to achieve this. Following are different methods to achieve same using bitwise operators.

**Method 1**  
To add 1 to a number x (say 0011000111), we need to flip all the bits after the rightmost 0 bit (we get 001100**0**000). Finally, flip the rightmost 0 bit also (we get 0011001000) and we are done.

|  |
| --- |
| #include<stdio.h>    int addOne(int x)  {    int m = 1;      /\* Flip all the set bits until we find a 0 \*/    while( x & m )    {      x = x^m;      m <<= 1;    }      /\* flip the rightmost 0 bit \*/    x = x^m;    return x;  }    /\* Driver program to test above functions\*/  int main()  {    printf("%d", addOne(13));    getchar();    return 0;  } |

Run on IDE

**Method 2**  
We know that the negative number is represented in 2’s complement form on most of the architectures. We have the following lemma hold for 2’s complement representation of signed numbers.

Say, x is numerical value of a number, then

~x = -(x+1) [ ~ is for bitwise complement ]

(x + 1) is due to addition of 1 in 2’s complement conversion

To get (x + 1) apply negation once again. So, the final expression becomes (-(~x)).

|  |
| --- |
| int addOne(int x)  {     return (-(~x));  }    /\* Driver program to test above functions\*/  int main()  {    printf("%d", addOne(13));    getchar();    return 0;  } |

Run on IDE

Example, assume the machine word length is one \*nibble\* for simplicity.  
And x = 2 (0010),  
~x = ~2 = 1101 (13 numerical)  
-~x = -1101  
Interpreting bits 1101 in 2’s complement form yields numerical value as -(2^4 – 13) = -3. Applying ‘-‘ on the result leaves 3. Same analogy holds for decrement. See [this](http://www.geeksforgeeks.org/archives/8198/comment-page-1#comment-2159)comment for implementation of decrement.  
Note that this method works only if the numbers are stored in 2’s complement form.

# **Multiply a given Integer with 3.5**

Given a integer x, write a function that multiplies x with 3.5 and returns the integer result. You are not allowed to use %, /, \*.

Examples:  
Input: 2  
Output: 7

Input: 5  
Output: 17 (Ignore the digits after decimal point)

Solution:  
**1.**We can get x\*3.5 by adding 2\*x, x and x/2. To calculate 2\*x, left shift x by 1 and to calculate x/2, right shift x by 2.

|  |
| --- |
| #include <stdio.h>    int multiplyWith3Point5(int x)  {    return (x<<1) + x + (x>>1);  }    /\* Driver program to test above functions\*/  int main()  {    int x = 4;    printf("%d", multiplyWith3Point5(x));    getchar();    return 0;  } |

Run on IDE

**2.**Another way of doing this could be (8\*x – x)/2 (See below code). Thanks to [ajaym](http://www.geeksforgeeks.org/archives/8210/comment-page-1#comment-1616)for suggesting this.

|  |
| --- |
| #include <stdio.h>  int multiplyWith3Point5(int x)  {    return ((x<<3) - x)>>1;  } |

# **Turn off the rightmost set bit**

Write a C function 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)

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.

|  |
| --- |
| #include<stdio.h>    /\* unsets the rightmost set bit of n and returns the result \*/  int fun(unsigned int n)  {    return n&(n-1);  }    /\* Driver program to test above function \*/  int main()  {    int n = 7;    printf("The number after unsetting the rightmost set bit %d", fun(n));      getchar();    return 0;  } |

# **Find whether a given number is a power of 4 or not**

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/probfunc-page.php?pid=700129)

**1.**A simple method is to take log of the given number on base 4, and if we get an integer then number is power of 4.

**2.**Another solution is to keep dividing the number by 4, i.e, do n = n/4 iteratively. In any iteration, if n%4 becomes non-zero and n is not 1 then n is not a power of 4, otherwise n is a power of 4.

|  |
| --- |
| #include<stdio.h>  #define bool int    /\* Function to check if x is power of 4\*/  bool isPowerOfFour(int n)  {    if(n == 0)      return 0;    while(n != 1)    {     if(n%4 != 0)        return 0;      n = n/4;    }    return 1;  }    /\*Driver program to test above function\*/  int main()  {    int test\_no = 64;    if(isPowerOfFour(test\_no))      printf("%d is a power of 4", test\_no);    else      printf("%d is not a power of 4", test\_no);    getchar();  } |

Run on IDE

**3.**A number n is a power of 4 if following conditions are met.  
a) There is only one bit set in the binary representation of n (or n is a power of 2)  
b) The count of zero bits before the (only) set bit is even.

For example: 16 (10000) is power of 4 because there is only one bit set and count of 0s before the set bit is 4 which is even.

Thanks to [Geek4u](http://www.geeksforgeeks.org/forum/topic/power-question#post-312)for suggesting the approach and providing the code.

|  |
| --- |
| #include<stdio.h>  #define bool int    bool isPowerOfFour(unsigned int n)  {    int count = 0;      /\*Check if there is only one bit set in n\*/    if ( n && !(n&(n-1)) )    {       /\* count 0 bits before set bit \*/       while(n > 1)       {         n  >>= 1;         count += 1;       }        /\*If count is even then return true else false\*/      return (count%2 == 0)? 1 :0;    }      /\* If there are more than 1 bit set      then n is not a power of 4\*/    return 0;  }    /\*Driver program to test above function\*/  int main()  {     int test\_no = 64;     if(isPowerOfFour(test\_no))       printf("%d is a power of 4", test\_no);     else       printf("%d is not a power of 4", test\_no);     getchar();  } |

# **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.

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

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**.

|  |
| --- |
| #include<stdio.h>  #define INT\_BITS 32    /\*Function to left rotate n by d bits\*/  int leftRotate(int n, unsigned int 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\*/  int rightRotate(int n, unsigned int 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));  }    /\* Driver program to test above functions \*/  int main()  {    int n = 16;    int d = 2;    printf("Left Rotation of %d by %d is ", n, d);    printf("%d", leftRotate(n, d));    printf("\nRight Rotation of %d by %d is ", n, d);    printf("%d", rightRotate(n, d));    getchar();  } |

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

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

**Example:**  
I/P = [1, 2, 3, 2, 3, 1, 3]  
O/P = 3

## [Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/find-the-odd-occurence/0)

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

**Program :**

|  |
| --- |
| // Java program to find the element occurring  // odd number of times  class OddOccurrence {        // funtion to find the element occurring odd      // number of times      static int getOddOccurrence(int arr[], int arr\_size)      {          int i;          for (i = 0; i < arr\_size; i++) {              int count = 0;              for (int j = 0; j < arr\_size; j++) {                  if (arr[i] == arr[j])                      count++;              }              if (count % 2 != 0)                  return arr[i];          }          return -1;      }        // driver code      public static void main(String[] args)      {          int arr[] = new int[]{ 2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2 };          int n = arr.length;          System.out.println(getOddOccurrence(arr, n));      }  }  // This code has been contributed by Kamal Rawal |

Run on IDE

Output :

5

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

# **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<stdio.h>  #include<stdlib.h>    /\* 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;   }     int main()   {       int \*res = (int \*)malloc(sizeof(int));       int x = 2147483640;       int y = 10;         printf("%d", addOvf(res, x, y));         printf("\n %d", \*res);       getchar();       return 0;  } |

Run on IDE

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.

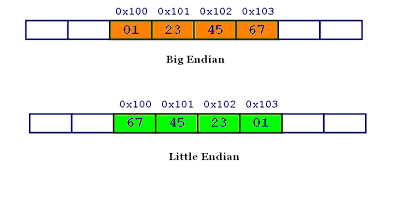
|  |
| --- |
| #include<stdio.h>  #include<limits.h>  #include<stdlib.h>    int addOvf(int\* result, int a, int b)  {     if( a > INT\_MAX - b)       return -1;     else     {       \*result = a + b;        return 0;     }  }    int main()  {    int \*res = (int \*)malloc(sizeof(int));    int x = 2147483640;    int y = 10;      printf("%d", addOvf(res, x, y));    printf("\n %d", \*res);    getchar();    return 0;  } |

Run on IDE

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

# **Little and Big Endian Mystery**

**What are these?**  
Little and big endian are two ways of storing multibyte data-types ( int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.  
  
Suppose integer is stored as 4 bytes (For those who are using DOS based compilers such as C++ 3.0 , integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.

[](http://4.bp.blogspot.com/_IEmaCFe3y9g/SO3GGEF4UkI/AAAAAAAAAAc/z7waF2Lwg0s/s1600-h/lb.GIF)

Memory representation of integer ox01234567 inside Big and little endian machines

**How to see memory representation of multibyte data types on your machine?**  
Here is a sample C code that shows the byte representation of int, float and pointer.

|  |
| --- |
| #include <stdio.h>    /\* function to show bytes in memory, from location start to start+n\*/  void show\_mem\_rep(char \*start, int n)  {      int i;      for (i = 0; i < n; i++)           printf(" %.2x", start[i]);      printf("\n");  }    /\*Main function to call above function for 0x01234567\*/  int main()  {     int i = 0x01234567;     show\_mem\_rep((char \*)&i, sizeof(i));     getchar();     return 0;  } |

Run on IDE

When above program is run on little endian machine, gives “67 45 23 01” as output , while if it is run on endian machine, gives “01 23 45 67” as output.

**Is there a quick way to determine endianness of your machine?**  
There are n no. of ways for determining endianness of your machine. Here is one quick way of doing the same.

|  |
| --- |
| #include <stdio.h>  int main()  {     unsigned int i = 1;     char \*c = (char\*)&i;     if (\*c)         printf("Little endian");     else         printf("Big endian");     getchar();     return 0;  } |

Run on IDE

In the above program, a character pointer c is pointing to an integer i. Since size of character is 1 byte when the character pointer is de-referenced it will contain only first byte of integer. If machine is little endian then \*c will be 1 (because last byte is stored first) and if machine is big endian then \*c will be 0.  
 **Does endianness matter for programmers?**  
Most of the times compiler takes care of endianness, however, endianness becomes an issue in following cases.

It matters in network programming: Suppose you write integers to file on a little endian machine and you transfer this file to a big endian machine. Unless there is little andian to big endian transformation, big endian machine will read the file in reverse order. You can find such a practical example here.

Standard byte order for networks is big endian, also known as network byte order. Before transferring data on network, data is first converted to network byte order (big endian).

Sometimes it matters when you are using type casting, below program is an example.

|  |
| --- |
| #include <stdio.h>  int main()  {      unsigned char arr[2] = {0x01, 0x00};      unsigned short int x = \*(unsigned short int \*) arr;      printf("%d", x);      getchar();      return 0;  } |

Run on IDE

In the above program, a char array is typecasted to an unsigned short integer type. When I run above program on little endian machine, I get 1 as output, while if I run it on a big endian machine I get 256. To make programs endianness independent, above programming style should be avoided.  
 **What are bi-endians?**  
Bi-endian processors can run in both modes little and big endian.

**What are the examples of little, big endian and bi-endian machines ?**  
Intel based processors are little endians. ARM processors were little endians. Current generation ARM processors are bi-endian.

Motorola 68K processors are big endians. PowerPC (by Motorola) and SPARK (by Sun) processors were big endian. Current version of these processors are bi-endians.  
 **Does endianness effects file formats?**  
File formats which have 1 byte as a basic unit are independent of endianness e..g., ASCII files . Other file formats use some fixed endianness forrmat e.g, JPEG files are stored in big endian format.

**Which one is better — little endian or big endian**  
The term little and big endian came from Gulliver’s Travels by Jonathan Swift. Two groups could not agree by which end a egg should be opened -a-the little or the big. Just like the egg issue, there is no technological reason to choose one byte ordering convention over the other, hence the arguments degenerate into bickering about sociopolitical issues. As long as one of the conventions is selected and adhered to consistently, the choice is arbitrary.

# **Count set bits in an integer**

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

Examples

Input : n = 6

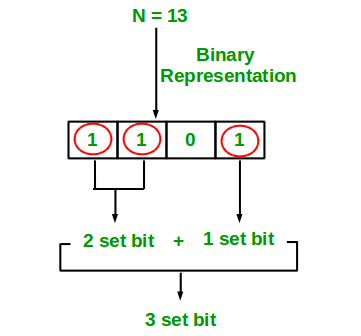
Output : 2

Binary representation of 6 is 110 and has 2 set bits

Input : n = 13

Output : 3

Binary representation of 11 is 1101 and has 3 set bits



## [Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/set-bits/0)

**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 below program.

|  |
| --- |
| #include <stdio.h>    /\* 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;      printf("%d", countSetBits(i));      return 0;  } |

Run on IDE

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

**2. Brian Kernighan’s Algorithm:**  
Subtraction of 1 from a number toggles all the bits (from right to left) till the rightmost set bit(including the righmost set bit). So if we subtract a number by 1 and do bitwise & with itself (n & (n-1)), we unset the righmost set bit. If we do n & (n-1) in a loop and count the no of times loop executes we get the set bit count.  
Beauty of the this solution is 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:**

|  |
| --- |
| #include<stdio.h>    /\* Function to get no of set bits in binary     representation of passed binary no. \*/  unsigned int countSetBits(int n)  {      unsigned int count = 0;      while (n)      {        n &= (n-1) ;        count++;      }      return count;  }    /\* Program to test function countSetBits \*/  int main()  {      int i = 9;      printf("%d", countSetBits(i));      getchar();      return 0;  } |

Run on IDE

**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)

**3. Using Lookup table:**We can count bits in O(1) time using lookup table. Please see<http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable> for details.

We can find one use of counting set bits at <http://www.geeksforgeeks.org/?p=1465>

**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;  } |

Run on IDE

Output :

1

4

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

Given two numbers ‘a’ and b’. Write a program to count number of bits needed to be flipped to convert ‘a’ to ‘b’.

Example :

Input : a = 10, b = 20

Output : 4

Binary representation of a is 000**0101**0

Binary representation of b is 00010100

We need to flip highlighted four bits in a

to make it b.

Input : a = 7, b = 10

Output : 3

Binary representation of a is 0000**01**1**1**

Binary representation of b is 00001010

We need to flip highlighted three bits in a

to make it b.

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/bit-difference/0)

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 covert A into B  #include <iostream>  using namespace std;    // Function that count set bits  int countSetBits(int n)  {      int count = 0;      while (n)      {          count += n & 1;          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;  } |

Run on IDE

Output :

4

# **Efficient way to multiply with 7**

We can multiply a number by 7 using bitwise operator. First left shift the number by 3 bits (you will get 8n) then subtract the original numberfrom the shifted number and return the difference (8n – n).  
  
 **Program:**

|  |
| --- |
| # include<stdio.h>    int multiplyBySeven(unsigned int n)  {      /\* Note the inner bracket here. This is needed         because precedence of '-' operator is higher         than '<<' \*/      return ((n<<3) - n);  }    /\* Driver program to test above function \*/  int main()  {      unsigned int n = 4;      printf("%u", multiplyBySeven(n));        getchar();      return 0;  } |

Run on IDE

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

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

Given a positive integer, write a function to find if it is a power of two or not.

Examples:

Input : n = 4

Output : Yes

22 = 4

Input : n = 7

Output : No

Input : n = 32

Output : Yes

25 = 32

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/pattern-jumping/0)

**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 number is power of 2.  
  
**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<stdio.h>  #define bool int    /\* 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 program to test above function\*/  int main()  {    isPowerOfTwo(31)? printf("Yes\n"): printf("No\n");    isPowerOfTwo(17)? printf("Yes\n"): printf("No\n");    isPowerOfTwo(16)? printf("Yes\n"): printf("No\n");    isPowerOfTwo(2)? printf("Yes\n"): printf("No\n");    isPowerOfTwo(18)? printf("Yes\n"): printf("No\n");    isPowerOfTwo(1)? printf("Yes\n"): printf("No\n");    return 0;  } |

Run on IDE

Output:

No

No

Yes

Yes

No

Yes

**3.**All power of two numbers have only one bit set. So count the no. of set bits and if you get 1 then number is a power of 2. Please see <http://www.geeksforgeeks.org/?p=1176> for counting set bits.

**4.**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 become unset.

For example for 4 ( 100) and 16(10000), we get 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 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 [Mohammad](http://www.geeksforgeeks.org/?p=535#comment-177)for adding this case).  
Below is the implementation of this method.

# **Position of rightmost set bit**

Write a one line C function to return position of first 1 from right to left, in binary representation of an Integer.

I/P 18, Binary Representation 010010

O/P 2

I/P 19, Binary Representation 010011

O/P 1

## [Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/find-first-set-bit/0)

**Algorithm:** (Example 18(010010))

Let I/P be 12 (1100)

1. Take two's complement of the given no as all bits are reverted

except the first '1' from right to left (10111)

2 Do an bit-wise & with original no, this will return no with the

required one only (00010)

3 Take the log2 of the no, you will get position -1 (1)

4 Add 1 (2)

**Program:**

* C
* Java

|  |
| --- |
| #include<stdio.h>  #include<math.h>    unsigned int getFirstSetBitPos(int n)  {     return log2(n&-n)+1;  }    int main()  {      int n = 12;      printf("%u", getFirstSetBitPos(n));      getchar();      return 0;  } |

Run on IDE

Output:

3

# **Binary representation of a given number**

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

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/binary-representation/0)

Source: [Microsoft Interview Set-3](http://www.geeksforgeeks.org/archives/23422)

**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.

|  |
| --- |
| void bin(unsigned n)  {      unsigned i;      for (i = 1 << 31; i > 0; i = i / 2)          (n & i)? printf("1"): printf("0");  }    int main(void)  {      bin(7);      printf("\n");      bin(4);  } |

Run on IDE

**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.

|  |
| --- |
| void bin(unsigned n)  {      /\* step 1 \*/      if (n > 1)          bin(n/2);        /\* step 2 \*/      printf("%d", n % 2);  }    int main(void)  {      bin(7);      printf("\n");      bin(4);  } |

# **Find position of the only set bit**

Given a number having only one ‘1’ and all other ’0’s in its binary representation, find position of the only set bit. Source: [Microsoft Interview | 18](http://www.geeksforgeeks.org/microsoft-interview-178/)

## [Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/find-position-of-set-bit/0)

The idea is to start from rightmost bit and one by one check value of every bit. Following is 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 given number is power of 2 or not. If given number is not 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 <stdio.h>    // A utility function to check whether n is 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)? printf("n = %d, Invalid numbern", n):                   printf("n = %d, Position %d n", n, pos);        n = 12;      pos = findPosition(n);      (pos == -1)? printf("n = %d, Invalid numbern", n):                   printf("n = %d, Position %d n", n, pos);        n = 128;      pos = findPosition(n);      (pos == -1)? printf("n = %d, Invalid numbern", n):                   printf("n = %d, Position %d n", n, pos);        return 0;  } |

Run on IDE

Output:

n = 16, Position 5

n = 12, Invalid number

n = 128, Position 8

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 position of the set bit.

|  |
| --- |
| // C program to find position of only set bit in a given number  #include <stdio.h>    // 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 program to test above function  int main(void)  {      int n = 0;      int pos = findPosition(n);      (pos == -1)? printf("n = %d, Invalid numbern", n):                   printf("n = %d, Position %d n", n, pos);        n = 12;      pos = findPosition(n);      (pos == -1)? printf("n = %d, Invalid numbern", n):                   printf("n = %d, Position %d n", n, pos);        n = 128;      pos = findPosition(n);      (pos == -1)? printf("n = %d, Invalid numbern", n):                   printf("n = %d, Position %d n", n, pos);        return 0;  } |

Run on IDE

Output:

n = 0, Invalid number

n = 12, Invalid number

n = 128, Position 8

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

|  |
| --- |
| #include <stdio.h>    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 program to test above function  int main(void)  {      int n = 0;      int pos = findPosition(n);      (pos == -1)? printf("n = %d, Invalid numbern", n):                   printf("n = %d, Position %d n", n, pos);        n = 12;      pos = findPosition(n);      (pos == -1)? printf("n = %d, Invalid numbern", n):                   printf("n = %d, Position %d n", n, pos);        n = 128;      pos = findPosition(n);      (pos == -1)? printf("n = %d, Invalid numbern", n):                   printf("n = %d, Position %d n", n, pos);        return 0;  } |

Run on IDE

Output:

n = 0, Invalid number

n = 12, Invalid number

n = 128, Position 8

# **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 sum in one of the two given numbers. The numbers can then be swapped using the sum and subtraction from sum.

|  |
| --- |
| #include <stdio.h>  int main()  {    int x = 10, y = 5;      // Code to swap 'x' and 'y'    x = x + y;  // x now becomes 15    y = x - y;  // y becomes 10    x = x - y;  // x becomes 5      printf("After Swapping: x = %d, y = %d", x, y);      return 0;  } |

Run on IDE

Output:

After Swapping: x = 5, y = 10

Multiplication and division can also be used for swapping.

|  |
| --- |
| #include <stdio.h>  int main()  {    int x = 10, y = 5;      // Code to swap 'x' and 'y'    x = x \* y;  // x now becomes 50    y = x / y;  // y becomes 10    x = x / y;  // x becomes 5      printf("After Swapping: x = %d, y = %d", x, y);      return 0;  } |

Run on IDE

Output:

After Swapping: x = 5, y = 10

**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 which 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).

|  |
| --- |
| #include <stdio.h>  int main()  {    int x = 10, y = 5;      // Code to swap 'x' (1010) and 'y' (0101)    x = x ^ y;  // x now becomes 15 (1111)    y = x ^ y;  // y becomes 10 (1010)    x = x ^ y;  // x becomes 5 (0101)      printf("After Swapping: x = %d, y = %d", x, y);      return 0;  } |

Run on IDE

Output:

After Swapping: x = 5, y = 10

**Problems with above methods**  
**1)** The multiplication and division based approach doesn’ work if one of the numbers is 0 as the product becomes 0 irrespective of the other number.

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

**3)** When we use pointers to variable and make a function swap, all of the above methods fail when both pointers point to the same variable. Let’s take a look 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.

|  |
| --- |
| #include <stdio.h>  void swap(int \*xp, int \*yp)  {      \*xp = \*xp ^ \*yp;      \*yp = \*xp ^ \*yp;      \*xp = \*xp ^ \*yp;  }    int main()  {    int x = 10;    swap(&x, &x);    printf("After swap(&x, &x): x = %d", x);    return 0;  } |

Run on IDE

Output:

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

Swapping a variable with itself may 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 the swapping.

|  |
| --- |
| #include <stdio.h>  void swap(int \*xp, int \*yp)  {      if (xp == yp) // Check if the two addresses are same        return;      \*xp = \*xp + \*yp;      \*yp = \*xp - \*yp;      \*xp = \*xp - \*yp;  }  int main()  {    int x = 10;    swap(&x, &x);    printf("After swap(&x, &x): x = %d", x);    return 0;  } |

Run on IDE

Output:

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

# **Swap two nibbles in a byte**

A [nibble](http://en.wikipedia.org/wiki/Nibble)is a four-bit aggregation, or half an octet. There are two nibbles in a byte.  
Given a byte, swap the two nibbles in it. For example 100 is be represented as 01100100 in a byte (or 8 bits). The two nibbles are (0110) and (0100). If we swap the two nibbles, we get 01000110 which is 70 in decimal.

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/swap-two-nibbles-in-a-byte/0)

To swap the nibbles, we can use bitwise &, bitwise ‘<<' and '>>’ operators. A byte can be represented using a unsigned char in C as size of char is 1 byte in a typical C compiler. Following is C program to swap the two nibbles in a byte.

|  |
| --- |
| #include <stdio.h>    unsigned char swapNibbles(unsigned char x)  {      return ( (x & 0x0F)<<4 | (x & 0xF0)>>4 );  }    int main()  {      unsigned char x = 100;      printf("%u", swapNibbles(x));      return 0;  } |

Run on IDE

Output:

70

**Explanation:**  
100 is 01100100 in binary. The operation can be split mainly in two parts  
**1)** The expression “**x & 0x0F**” gives us last 4 bits of x. For x = 100, the result is 00000100. Using bitwise ‘<<' operator, we shift the last four bits to the left 4 times and make the new last four bits as 0. The result after shift is 01000000. **2)** The expression “**x & 0xF0**” gives us first four bits of x. For x = 100, the result is 01100000. Using bitwise ‘>>’ operator, we shift the digit to the right 4 times and make the first four bits as 0. The result after shift is 00000110.

At the end we use the bitwise OR ‘|’ operation of the two expressions explained above. The OR operator places first nibble to the end and last nibble to first. For x = 100, the value of (01000000) OR (00000110) gives the result 01000110 which is equal to 70 in decimal.

# **How to turn off a particular bit in a number?**

**Difficulty Level:** Rookie

Given a number n and a value k, turn of the k’th bit in n.

Examples:

Input: n = 15, k = 1

Output: 14

Input: n = 15, k = 2

Output: 13

Input: n = 15, k = 3

Output: 11

Input: n = 15, k = 4

Output: 7

Input: n = 15, k >= 5

Output: 15

The idea is to use bitwise <<, & and ~ operators. Using expression "**~(1 << (k - 1))**“, we get a number which has all bits set, except the k’th bit. If we do bitwise & of this expression with n, we get a number which has all bits same as n except the k’th bit which is 0.

Following is C++ implementation of this.

* C/C++
* Java

|  |
| --- |
| #include <iostream>  using namespace std;    // Returns a number that has all bits same as n  // except the k'th bit which is made 0  int turnOffK(int n, int k)  {      // k must be greater than 0      if (k <= 0) return n;        // Do & of n with a number with all set bits except      // the k'th bit      return (n & ~(1 << (k - 1)));  }    // Driver program to test above function  int main()  {      int n = 15;      int k = 4;      cout << turnOffK(n, k);      return 0;  } |

Run on IDE

Output:

7

**Exercise:** Write a function turnOnK() that turns the k’th bit on.

# **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](http://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'.

* C/C++
* Java

|  |
| --- |
| #include <iostream>  using namespace std;    // A method to multiply two numbers using Russian Peasant method  unsigned int russianPeasant(unsigned int a, unsigned int b)  {      int 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  int main()  {      cout << russianPeasant(18, 1) << endl;      cout << russianPeasant(20, 12) << endl;      return 0;  } |

Run on IDE

Output:

18

240

**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.

# **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.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=569)

Since sizes of two strings may be different, we first make the size of smaller string equal to that of bigger string by adding leading 0s. After making sizes 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 begining 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 C++ implementation of the above algorithm.

|  |
| --- |
| #include <iostream>  using namespace std;    //adds the two bit strings and return the result  string addBitStrings( string first, string second );    // Helper method: given two unequal sized bit strings, converts them to  // same length by aadding leading 0s in the smaller string. Returns the  // the new length  int makeEqualLength(string &str1, string &str2)  {      int len1 = str1.size();      int len2 = str2.size();      if (len1 < len2)      {          for (int i = 0 ; i < len2 - len1 ; i++)              str1 = '0' + str1;          return len2;      }      else if (len1 > len2)      {          for (int i = 0 ; i < len1 - len2 ; i++)              str2 = '0' + str2;      }      return len1; // If len1 >= len2  }    // The main function that adds two bit sequences and returns the addition  string addBitStrings( string first, string second )  {      string result;  // To store the sum bits        // make the lengths same before adding      int length = makeEqualLength(first, second);        int carry = 0;  // Initialize carry        // Add all bits one by one      for (int i = length-1 ; i >= 0 ; i--)      {          int firstBit = first.at(i) - '0';          int secondBit = second.at(i) - '0';            // boolean expression for sum of 3 bits          int sum = (firstBit ^ secondBit ^ carry)+'0';            result = (char)sum + result;            // boolean expression for 3-bit addition          carry = (firstBit & secondBit) | (secondBit & carry) | (firstBit & carry);      }        // if overflow, then add a leading 1      if (carry)          result = '1' + result;        return result;  }    // Driver program to test above functions  int main()  {      string str1 = "1100011";      string str2 = "10";        cout << "Sum is " << addBitStrings(str1, str2);      return 0;  } |

Run on IDE

Output:

Sum is 1100101

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

Following are not allowed to use  
1) Arithmetic and Comparison Operators  
2) String functions

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

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

|  |
| --- |
| #include <iostream>  using namespace std;    void areSame(int a, int b)  {     if (a^b) cout << "Not Same";     else cout << "Same";  }    int main()  {     areSame(10, 20);  } |

Run on IDE

Output:

Not Same

Source: <http://www.geeksforgeeks.org/count-of-n-digit-numbers-whose-sum-of-digits-equals-to-given-sum/>

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

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

Given two integers, find XOR of them without using 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.](http://ide.geeksforgeeks.org/)

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

|  |
| --- |
| // C++ program to find XOR without using ^  #include <iostream>  using namespace std;    // Returns XOR of x and y  int myXOR(int x, int y)  {      int res = 0; // Initialize result        // Assuming 32-bit Integer      for (int i = 31; i >= 0; i--)      {         // Find current bits in x and y         bool b1 = x & (1 << i);         bool b2 = y & (1 << i);            // If both are 1 then 0 else xor is same as OR          bool xoredBit = (b1 & b2) ? 0 : (b1 | b2);            // Update result          res <<= 1;          res |= xoredBit;      }      return res;  }    // Driver program to test above function  int main()  {     int x = 3, y = 5;     cout << "XOR is " << myXOR(x, y);     return 0;  } |

Run on IDE

Output:

XOR is 6

Thanks to [Utkarsh Trivedi](http://qa.geeksforgeeks.org/user/utkarsh111) for suggesting this solution.

A **Better Solution** can find XOR without using 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 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 C++ implementation.

|  |
| --- |
| // C++ program to find XOR without using ^  #include <iostream>  using namespace std;    // Returns XOR of x and y  int myXOR(int x, int y)  {     return (x | y) & (~x | ~y);  }    // Driver program to test above function  int main()  {     int x = 3, y = 5;     cout << "XOR is " << myXOR(x, y);     return 0;  } |

Run on IDE

Output:

XOR is 6

# **XOR counts of 0s and 1s in binary representation**

Given a number, the task is to find XOR of count of 0s and count of 1s in binary representation of a given number.  
Examples:

Input : 5

Output : 3

Binary representation : 101

Count of 0s = 1,

Count of 1s = 2

1 XOR 2 = 3.

Input : 7

Output : 3

Binary representation : 111

Count of 0s = 0

Count of 1s = 3

0 XOR 3 = 3.

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

The idea is simple, we traverse through all bits of a number, count 0s and 1s and finally return XOR of two counts.

// C++ program to find XOR of counts 0s and 1s in

// binary representation of n.

#include<iostream>

using namespace std;

// Returns XOR of counts 0s and 1s in

// binary representation of n.

int countXOR(int n)

{

int count0 = 0, count1 = 0;

while (n)

{

//calculating count of zeros and ones

(n % 2 == 0) ? count0++ :count1++;

n /= 2;

}

return (count0 ^ count1);

}

// Driver Program

int main()

{

int n = 31;

cout << countXOR (n);

return 0;

}

Output:

5

One observation is, for a number of the form **2^x – 1**, the output is always x. We can directly produce answer for this case by first checking n+1 is a[power of two or not](http://www.geeksforgeeks.org/write-one-line-c-function-to-find-whether-a-no-is-power-of-two/).

# **Multiply a number with 10 without using multiplication operator**

Given a number, the task is to multiply it with 10 without using multiplication operator?

Examples:

Input : n = 50

Output: 500

// multiplication of 50 with 10 is = 500

Input : n = 16

Output: 160

// multiplication of 16 with 10 is = 160

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

A **simple solution** for this problem is to run a loop and add n with itself 10 times. Here we need to perform 10 operations.  
A **better solution** is to use bit manipulation. We have to multiply n with 10 i.e; n\*10, we can write this as **n\*(2+8) = n\*2 + n\*8** and since we are not allowed to use multiplication operator we can do this using left shift bitwise operator. So n\*10 = n<<1 + n<<3.

* C++
* Java

|  |
| --- |
| // C++ program to multiply a number with 10 using  // bitwise operators  #include<bits/stdc++.h>  using namespace std;    // Function to find multiplication of n with  // 10 without usng multiplication operator  int multiplyTen(int n)  {      return (n<<1) + (n<<3);  }    // Driver program to run the case  int main()  {      int n = 50;      cout << multiplyTen(n);      return 0;  } |

Run on IDE

Output:

500

# **Equal Sum and XOR**

Given a positive integer n, find count of positive integers i such that 0 <= i <= n and n+i = n^i

Input : n = 7

Output : 1

Explanation:

7^i = 7+i holds only for only for i = 0

7+0 = 7^0 = 7

Input n = 12

Output: 4

12^i = 12+i hold only for i = 0, 1, 2, 3

for i=0, 12+0 = 12^0 = 12

for i=1, 12+1 = 12^1 = 13

for i=2, 12+2 = 12^2 = 14

for i=3, 12+3 = 12^3 = 15

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/equal-sum-and-xor/1)

**Method 1 (Simple) :**

One simple solution is to iterate over all values of i 0<= i <= n and count all satisfying values.

* C++
* Java

|  |
| --- |
| /\* C++ program to print count of values such     that n+i = n^i \*/  #include <iostream>  using namespace std;    // function to count number of values less than  // equal to n that satisfy the given condition  int countValues (int n)  {      int countV = 0;        // Traverse all numbers from 0 to n and      // increment result only when given condition      // is satisfied.      for (int i=0; i<=n; i++ )          if ((n+i) == (n^i) )              countV++;        return countV;  }    // Driver program  int main()  {      int n = 12;      cout << countValues(n);      return 0;  } |

Run on IDE

Output:

4

**Method 2 (Efficient) :**

An efficient solution is as follows

Since we know a + b = a ^ b + a & b

We can write, n + i = n ^ i + n & i

So n + i = n ^ i implies n & i = 0

Hence our problem reduces to finding values of i such that n & i = 0. How to find count of such pairs? We can use the count of unset-bits in the binary representation of n. For n & i to be zero, i must unset all set-bits of n. If the kth bit is set at a particular in n, kth bit in i must be 0 always, else kth bit of i can be 0 or 1

Hence, total such combinations are 2^(count of unset bits in n)

For example, consider n = 12 (Binary representation : 1 1 0 0).  
All possible values of i that can unset all bits of n are 0 0 0/1 0/1 where 0/1 implies either 0 or 1. Number of such values of i are 2^2 = 4.

The following is the program following the above idea.

* C++
* Java

|  |
| --- |
| /\* c++ program to print count of values such    that n+i = n^i \*/  #include <bits/stdc++.h>  using namespace std;    // function to count number of values less than  // equal to n that satisfy the given condition  int countValues(int n)  {      // unset\_bits keeps track of count of un-set      // bits in binary representation of n      int unset\_bits=0;      while (n)      {          if ((n & 1) == 0)              unset\_bits++;          n=n>>1;      }        // Return 2 ^ unset\_bits      return 1 << unset\_bits;  }    // Driver code  int main()  {      int n = 12;      cout << countValues(n);      return 0;  } |

Run on IDE

Output:

4

# **Swap three variables without using temporary variable**

Given three variables, a, b and c, swap them without temporary variable.

Input : a = 10, b = 20 and c = 30

Output : a = 30, b = 10 and c = 20

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

**Method 1 (Using Arithmetic Operators)**

The idea is to get sum in one of the two given numbers. The numbers can then be swapped using the sum and subtraction from sum.  
We have already discussed swapping two variables [here](http://www.geeksforgeeks.org/swap-two-numbers-without-using-temporary-variable/). We can extend the same approaches

|  |
| --- |
| // C++ program to swap three variables  // without using temporary variable.  #include <iostream>  using namespace std;    // Assign c's value to a, a's value to b and  // b's value to c.  void swapThree(int &a, int &b, int &c)  {      // Store sum of all in a      a = a + b + c;  // (a = 60)        // After this, b has value of a      b = a - (b+c);  // (b = 60 – (20+30) =10)        // After this, c has value of b      c = a - (b+c);  // (c = 60 – (10 + 30) = 20)        // After this, a has value of c      a = a - (b+c);   //(a = 60 – (10 + 20) = 30)  }    // Driver code  int main()  {      int a = 10, b = 20, c = 30;        cout << "Before swapping a = " << a << ", b = "           << b << ", c = " << c << endl;        swapThree(a, b, c);        cout << "After swapping a = " << a << ", b = "           << b << ", c = " << c << endl;        return 0;  } |

Run on IDE

Output:

Before swapping a = 10, b = 20, c = 30

After swapping a = 30, b = 10, c = 20

Thanks to [Mazhar MIK](http://qa.geeksforgeeks.org/user/Mazhar+MIK) for suggesting this method [here](http://qa.geeksforgeeks.org/9501/swaping-of-three-variables-without-using-another-variable).

**Method 2 (Using Bitwise XOR)**

The bitwise XOR operator can be used to swap three variables. The idea is similar to method 1. We first store XOR of all numbers in ‘a’. Then we get individual numbers by doing XOR of this with other two numbers.

|  |
| --- |
| // C++ program to swap three variables  // without using temporary variable  #include <iostream>  using namespace std;    // Assign c's value to a, a's value to b and  // b's value to c.  void swapThree(int &a, int &b, int &c)  {      // Store XOR of all in a      a = a ^ b ^ c;        // After this, b has value of a      b = a ^ b ^ c;        // After this, c has value of b      c = a ^ b ^ c;        // After this, a has value of c      a = a ^ b ^ c;  }    // Driver code  int main()  {      int a = 10, b = 20, c = 30;        cout << "Before swapping a = " << a << ", b = "           << b << ", c = " << c << endl;        swapThree(a, b, c);        cout << "After swapping a = " << a << ", b = "           << b << ", c = " << c << endl;        return 0;  } |

Run on IDE

Output:

Before swapping a = 10, b = 20, c = 30

After swapping a = 30, b = 10, c = 20

The method 1 causes overflow for large values of a, b and c, while method 2 doesn’t.

# **Efficient method for 2’s complement of a binary string**

Given a Binary Number as string, print its 2’s complements.

**2’s complement**of a binary number is 1 added to the 1’s complement of the binary number. Note that 1’s complement is simply flip of given binary number.  
Examples:

2's complement of "0111" is "1001"

2's complement of "1100" is "0100"

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=250)

We have discussed 1’s and 2’s complements in below post.

[1’s and 2’s complement of a Binary Number.](http://www.geeksforgeeks.org/1s-2s-complement-binary-number/)

The method discussed in above post traverses binary string twice to find 2’s complement, first finds 1’s complement, then finds 2’s complement using 1’s complement

In this post an efficient method for 2’s complement is discussed that **traverses string only once**. We traverse the string from last till the single 1 is not traversed and after that flip all values of string i.e. 0 to 1 and 1 to 0.

**Note:**Here to handle the corner case i.e. if 1 doesn’t exist in the string then just append 1 in the starting of string.

**Illustration :**

**Input:**  str = "1000100"

**Output:**  0111100

Explanation: Starts traversing the string from last,

we got first '1' at index 4 then just flip the bits

of 0 to 3 indexes to make the 2's complement.

**Input:**  str = "0000"

**Output:**  10000

Explanation: As there is no 1 in the string so just

append '1' at starting.

* C++
* Java

|  |
| --- |
| // An efficient C++ program to find 2's complement  #include<bits/stdc++.h>  using namespace std;    // Function to find two's complement  string findTwoscomplement(string str)  {      int n = str.length();        // Traverse the string to get first '1' from      // the last of string      int i;      for (i = n ; i >= 0 ; i--)          if (str[i] == '1')              break;        // If there exists no '1' concat 1 at the      // starting of string      if (i == 0)          return '1' + str;        // Continue traversal after the position of      // first '1'      for (int k = i-1 ; k >= 0; k--)      {          //Just flip the values          if (str[k] == '1')              str[k] = '0';          else              str[k] = '1';      }        // return the modified string      return str;;  }    // Driver code  int main()  {      string str = "00000101";      cout << findTwoscomplement(str);      return 0;  } |

Run on IDE

Output:

11111011

# **Toggle case of a string using Bitwise operators**

Given a string, write a function that returns toggle case of a string using the bitwise operators in place.

In [ASCII](http://www.asciitable.com/) codes character ‘A’ is integer 65 = (0100 0001)2, while character ‘a’ is integer 97 = (0110 0001)2.

The difference between the ASCII values of ‘a’ and ‘A’ is 32.

So we can easily change the case of the letters either from Upper to lower or lower to upper by adding or subtracting the difference from the letters.

Examples:

Input : "GeekSfOrgEEKs"

Output : "gEEKsFoRGeekS"

Input : "StRinG"

Output : "sTrINg"

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

The[ASCII table](http://www.asciitable.com/)is constructed in such way that the binary representation of lowercase letters is almost identical of binary representation of uppercase letters.

**1. Lower case to CAPS**  
Bitwise XORing of a lowercase letter with a value of 65 (ASCII of ‘A’) or 90 (ASCII of ‘Z’) returns a value from 32 to 59, which upon bitwise ANDing with 32 always returns a nonzero of 32. So we convert it to uppercase.

**2. CAPS to Lower case**  
Bitwise XORing of a uppercase letter with a value of 65 (ASCII of ‘A’) or 90 (ASCII of ‘Z’) returns a value from 0 to 27, which upon bitwise ANDing with 32 always returns a zero. So we convert it to lowercase as above.

* C
* Java

|  |
| --- |
| // C program to get toggle case of a string  #include <stdio.h>  const int x=32;    // tOGGLE cASE = swaps CAPS to lower  // case and lower case to CAPS  char \*toggleCase(char \*a)  {      for (int i=0; a[i]!='\0'; i++){            // We can also do a[i]^90          if ((a[i]^65) & x)              a[i] &= ~x;          else              a[i] |= x;      }        return a;  }    // Driver Code  int main()  {      char str[] = "CheRrY";      printf("Toggle case: %s\n", toggleCase(str));      printf("Original string: %s", toggleCase(str));      return 0;  } |

Run on IDE

Output:

Toggle case: cHErRy

Original string: CheRrY

# **Toggling k-th bit of a number**

For a given number n, if k-th bit is 0, then toggle it to 1 and if it is 1 then, toggle it to 0.

Examples:

Input : n = 5, k = 1

Output : 4

5 is represented as 101 in binary

and has its first bit 1, so toggling

it will result in 100 i.e. 4.

Input : n = 2, k = 3

Output : 6

Input : n = 75, k = 4

Output : 67

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

Below are simple steps to find value of k-th bit

1) Left shift given number 1 by k-1 to create

a number that has only set bit as k-th bit.

temp = 1 << (k-1)

2) Return bitwise XOR of temp and n. Since temp

has only k-th bit set, doing XOR would toggle

only this bit.

**Example:**

n = 75 and k = 4

temp = 1 << (k-1) = 1 << 3 = 8

Binary Representation of temp = 0..00001000

Binary Representation of n = 0..0100**1**011

Bitwise XOR of two numbers = 0..0100**0**011

* C++
* Java

|  |
| --- |
| // CPP program to toggle k-th bit of n  #include<iostream>  using namespace std;    int toggleKthBit(int n, int k)  {      return (n ^ (1 << (k-1)));  }    // Driver code  int main()  {      int n = 5, k = 1;      cout << toggleKthBit(n , k);      return 0;  } |

Run on IDE

Output:

4

# **Convert decimal fraction to binary number**

Given an fraction decimal number n and integer k, convert decimal number n into equivalent binary number up-to k precision after decimal point.

Input: n = 2.47, k = 5

Output: 10.01111

Input: n = 6.986 k = 8

Output: 110.11111100

We strongly recommend that you click here and practice it, before moving on to the solution.

**A) Convert the integral part of decimal to binary equivalent**

1. Divide the decimal number by 2 and store remainders in array.
2. Divide the quotient by 2.
3. Repeat step 2 until we get the quotient equal to zero.
4. Equivalent binary number would be reverse of all remainders of step 1.

**B) Convert the fractional part of decimal to binary equivalent**

1. Multiply the fractional decimal number by 2.
2. Integral part of resultant decimal number will be first digit of fraction binary number.
3. Repeat step 1 using only fractional part of decimal number and then step 2.

**C) Combine both integral and fractional part of binary number.**

**Illustration**

Let's take an example for n = 4.47 k = 3

**Step 1: Conversion of 5 to binary**

1. 4/2 : Remainder = 0 : Quotient = 2

2. 2/2 : Remainder = 0 : Quotient = 1

3. 1/2 : Remainder = 1 : Quotient = 0

*So equivalent binary of integral part of decimal is 100.*

**Step 2: Conversion of .47 to binary**

1. 0.47 \* 2 = 0.94, Integral part: 0

2. 0.94 \* 2 = 1.88, Integral part: 1

3. 0.88 \* 2 = 1.76, Integral part: 1

*So equivalent bianry of fractional part of decimal is .011*

**Step 3: Combined the result of step 1 and 2.**

Final answer can be written as:

100 + .011 = 100.011

C++ Program to demonstrate above steps:

|  |
| --- |
| // C++ program to convert fractional decimal  // to binary number  #include<bits/stdc++.h>  using namespace std;    // Function to convert decimal to binary upto  // k-precision after decimal point  string decimalToBinary(double num, int k\_prec)  {      string binary = "";        // Fetch the integral part of decimal number      int Integral = num;        // Fetch the fractional part decimal number      double fractional = num - Integral;        // Conversion of integral part to      // binary equivalent      while (Integral)      {          int rem = Integral % 2;            // Append 0 in binary          binary.push\_back(rem +'0');            Integral /= 2;      }        // Reverse string to get original binary      // equivalent      reverse(binary.begin(),binary.end());        // Append point before conversion of      // fractional part      binary.push\_back('.');        // Conversion of fractional part to      // binary equivalent      while (k\_prec--)      {          // Find next bit in fraction          fractional \*= 2;          int fract\_bit = fractional;            if (fract\_bit == 1)          {              fractional -= fract\_bit;              binary.push\_back(1 + '0');          }          else              binary.push\_back(0 + '0');      }        return binary;  }    // Driver code  int main()  {        double n = 4.47;      int k = 3;      cout << decimalToBinary(n, k) << "\n";        n = 6.986 , k = 5;      cout << decimalToBinary(n, k);      return 0;  } |

Run on IDE

**Output:**

100.011

110.11111

**Time complexity:**O(len(n))  
**Auxiliary space:**O(len(n))

# **Toggle all the bits of a number except k-th bit.**

Given a positive (or unsigned) integer **n**, write a function to toggle all the bits except k-th bit. Here value of k starts from 0 (zero) and from right.

Examples:

Input : n = 4294967295, k = 0

Output : 1

The number 4294967295 in 32 bits has all bits

set. When we toggle all bits except last bit,

we get 1.

Input : n = 1, k = 1

Output : 4294967292

4294967262 has all bits toggled except second

bit from right.

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

1. Toggle bit at k-th position. We do it by finding a number with only k-th bit set (using 1 << k), then doing bitwise XOR of this number n.
2. Toggle all bits of number obtained above using ~ ([Bitwise negation](http://www.geeksforgeeks.org/interesting-facts-bitwise-operators-c/))

|  |
| --- |
| // C program to toggle all bits except kth bit  #include<stdio.h>    // Returns a number with all bit toggled in n  // except k-th bit  unsigned int toggleAllExceptK(unsigned int n,                              unsigned int k)  {     /\* 1) Toggle k-th bit by doing n ^ (1 << k)        2) Toggle all bits of the modified number \*/      return ~(n ^ (1 << k));  }    // Driver code  int main()  {      unsigned int n = 4294967295;      unsigned int k = 0;      printf("%u", toggleAllExceptK( n, k));      return 0;  } |

Run on IDE

Output:

1

# **Set the rightmost unset bit**

Given a non-negative number **n**. The problem is to set the rightmost unset bit in the binary representation of **n**. If there are no unset bits, then just leave the number as it is.

Examples:

Input : 21

Output : 23

**(21)10** = (101**0**1)2

Rightmost unset bit is at position **2**(from right) as

highlighted in the binary representation of **21**.

(23)10 = (101**1**1)2

The bit at position **2** has been set.

Input : 15

Output : 15

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

**Approach:** Following are the steps:

1. If **n** = 0, return 1.
2. If all bits of **n** are set, return n. Refer this post.
3. Else perform bitwise not on the given number(operation equivalent to 1’s complement). Let it be **num** = ~n.
4. Get the position of rightmost set bit of **num**. Let the position be **pos**.
5. Return **(1 << (pos – 1)) | n**.

|  |
| --- |
| // C++ implementation to set the rightmost unset bit  #include <bits/stdc++.h>  using namespace std;    // function to find the position  // of rightmost set bit  int getPosOfRightmostSetBit(int n)  {      return log2(n&-n)+1;  }    int setRightmostUnsetBit(int n)  {      // if n = 0, return 1      if (n == 0)          return 1;        // if all bits of 'n' are set      if ((n & (n + 1)) == 0)          return n;        // position of rightmost unset bit in 'n'      // passing ~n as argument      int pos = getPosOfRightmostSetBit(~n);        // set the bit at position 'pos'      return ((1 << (pos - 1)) | n);  }    // Driver program to test above  int main()  {      int n = 21;      cout << setRightmostUnsetBit(n);      return 0;  } |

Run on IDE

Output:

23

# **Convert a binary number to octal**

The problem is to convert the given binary number (represented as string) to its equivalent octal number. The input could be very large and may not fit even into unsigned long long int.

Examples:

Input : 110001110

Output : 616

Input : 1111001010010100001.010110110011011

Output : 1712241.26633

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

The idea is to consider the binary input as a string of characters and then follow the steps:

1. Get length of substring to the left and right of the decimal point(‘.’) as **left\_len** and **right\_len**.
2. If **left\_len** is not a multiple of 3 add min number of 0’s in the beginning to make length of left substring a multiple of 3.
3. If **right\_len** is not a multiple of 3 add min number of 0’s in the end to make length of right substring a multiple of 3.
4. Now, from the left extract one by one substrings of length 3 and add its corresponding octal code to the result.
5. If in between a decimal(‘.’) is encountered then add it to the result.

|  |
| --- |
| // C++ implementation to convert a binary number  // to octal number  #include <bits/stdc++.h>  using namespace std;    // function to create map between binary  // number and its equivalent octal  void createMap(unordered\_map<string, char> \*um)  {      (\*um)["000"] = '0';      (\*um)["001"] = '1';      (\*um)["010"] = '2';      (\*um)["011"] = '3';      (\*um)["100"] = '4';      (\*um)["101"] = '5';      (\*um)["110"] = '6';      (\*um)["111"] = '7';  }    // Function to find octal equivalent of binary  string convertBinToOct(string bin)  {      int l = bin.size();      int t = bin.find\_first\_of('.');        // length of string before '.'      int len\_left = t != -1 ? t : l;        // add min 0's in the beginning to make      // left substring length divisible by 3      for (int i = 1; i <= (3 - len\_left % 3) % 3; i++)          bin = '0' + bin;        // if decimal point exists      if (t != -1)      {          // length of string after '.'          int len\_right = l - len\_left - 1;            // add min 0's in the end to make right          // substring length divisible by 3          for (int i = 1; i <= (3 - len\_right % 3) % 3; i++)              bin = bin + '0';      }        // create map between binary and its      // equivalent octal code      unordered\_map<string, char> bin\_oct\_map;      createMap(&bin\_oct\_map);        int i = 0;      string octal = "";        while (1)      {          // one by one extract from left, substring          // of size 3 and add its octal code          octal += bin\_oct\_map[bin.substr(i, 3)];          i += 3;          if (i == bin.size())              break;            // if '.' is encountered add it to result          if (bin.at(i) == '.')          {              octal += '.';              i++;          }      }        // required octal number      return octal;  }    // Driver program to test above  int main()  {      string bin = "1111001010010100001.010110110011011";      cout << "Octal number = "           << convertBinToOct(bin);      return 0;  } |

Run on IDE

Output:

Octal number = 1712241.26633

# **Check in binary array the number represented by a subarray is odd or even**

Given a array such that all its terms is either 0 or 1.You need to tell the number represented by a subarray a[l..r] is odd or even

Examples:

Input : arr = {1, 1, 0, 1}

l = 1, r = 3

Output : odd

number represented by arr[l...r] is

101 which 5 in decimal form which is

odd

Input : arr = {1, 1, 1, 1}

l = 0, r = 3

Output : odd

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

The important point to note here is all the odd numbers in binary form have 1 as their rightmost bit and all even numbers have 0 as their rightmost bit.  
The reason is simple all other bits other than rightmost bit have even values and sum of even numbers is always even .Now the rightmost bit can have value either 1 or 0 as we know even + odd = odd so when right most bit is 1 the number is odd and when it is 0 the number is even.  
So to solve this problem we have to just check if a[r] is 0 or 1 and accordingly print odd or even

// C++ program to find if a subarray

// is even or odd.

#include<bits/stdc++.h>

using namespace std;

// prints if subarray is even or odd

void checkEVENodd (int arr[], int n, int l, int r)

{

// if arr[r] = 1 print odd

if (arr[r] == 1)

cout << "odd" << endl;

// if arr[r] = 0 print even

else

cout << "even" << endl;

}

// driver code

int main()

{

int arr[] = {1, 1, 0, 1};

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

checkEVENodd (arr, n, 1, 3);

return 0;

}

Output:

odd

# **Toggle the last m bits**

Given a non-negative number **n**. The problem is to toggle the last **m** bits in the binary representation of **n**. A **toggle** operation flips a bit **0** to **1** and a bit **1** to **0**.

**Constraint:** 1 <= m <= n.

Examples:

Input : n = 21, m = 2

Output : 22

**(21)10** = (101**01**)2

**(22)10** = (101**10**)2

The last two bits in the binary

representation of **21** are toggled.

Input : n = 107, m = 4

Output : 100

# **Toggle bits in the given range**

Given a non-negative number **n** and two values **l** and **r**. The problem is to toggle the bits in the range **l** to **r** in the binary representation of **n**, i.e, to toggle bits from the rightmost **lth** bit to the rightmost **rth**bit. A toggle operation flips a bit **0** to **1** and a bit **1** to **0**.

**Constraint:** 1 <= l <= r <= number of bits in the binary representation of **n**.

Examples:

Input : n = 17, l = 2, r = 3

Output : 23

**(17)10** = (10**00**1)2

**(23)10** = (10**11**1)2

The bits in the range **2** to **3** in the binary

representation of **17** are toggled.

Input : n = 50, l = 2, r = 5

Output : 44

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

**Approach:** Following are the steps:

1. Calculate **num** = ((1 << r) – 1) ^ ((1 << (l-1)) – 1). This will produce a number **num** having **r**number of bits and bits in the range **l** to **r** are the only set bits.
2. Now, perform **n = n ^ num**. This will toggle the bits in the range **l** to **r** in **n**.

* C/C++
* Java

|  |
| --- |
| // C++ implementation to toggle bits in  // the given range  #include <bits/stdc++.h>  using namespace std;    // function to toggle bits in the given range  unsigned int toggleBitsFromLToR(unsigned int n,                                  unsigned int l, unsigned int r)  {      // calculating a number 'num' having 'r'      // number of bits and bits in the range l      // to r are the only set bits      int num = ((1 << r) - 1) ^ ((1 << (l - 1)) - 1);        // toggle bits in the range l to r in 'n'      // and return the number      return (n ^ num);  }    // Driver program to test above  int main()  {      unsigned int n = 50;      unsigned int l = 2, r = 5;      cout << toggleBitsFromLToR(n, l, r);      return 0;  } |

Run on IDE

Output:

44

# **Unset bits in the given range**

Given a non-negative number **n** and two values **l** and **r**. The problem is to unset the bits in the range **l**to **r** in the binary representation of **n**, i.e, to unset bits from the rightmost **lth** bit to the rightmost **rth**bit.

**Constraint:** 1 <= l <= r <= number of bits in the binary representation of **n**.

Examples:

Input : n = 42, l = 2, r = 5

Output : 32

**(42)10** = (1**0101**0)2

**(32)10** = (1**0000**0)2

The bits in the range **2** to **5** in the binary

representation of **42** have been unset.

Input : n = 63, l = 1, r = 4

Output : 48

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

**Approach:** Following are the steps:

1. Calculate **num** = (1 << (sizeof(int) \* 8 – 1)) – 1. This will produce the highest positive integer **num**. All the bits in **num** will be set.
2. Toggle bits in the range **l** to **r** in **num**. Refer [this](http://www.geeksforgeeks.org/toggle-bits-given-range/) post.
3. Now, perform **n = n & num**. This will unset the bits in the range **l** to **r** in **n**.
4. Return **n**.

**Note:** The **sizeof(int)** has been used as input is of **int** data type. For large inputs you can use **long int** or **long long int** datatypes in place of **int**.

* C/C++
* Java

|  |
| --- |
| // C++ implementation to unset bits in the given range  #include <bits/stdc++.h>    using namespace std;    // function to toggle bits in the given range  unsigned int toggleBitsFromLToR(unsigned int n,                                  unsigned int l,                                  unsigned int r)  {      // calculating a number 'num' having 'r' number of bits      // and bits in the range l to r are the only set bits      int num = ((1 << r) - 1) ^ ((1 << (l - 1)) - 1);        // toggle the bits in the range l to r in 'n'      // and return the number      return (n ^ num);  }    // function to unset bits in the given range  unsigned int unsetBitsInGivenRange(unsigned int n,                                     unsigned int l, unsigned int r)  {      // 'num' is the highest positive integer number      // all the bits of 'num' are set      unsigned int num = (1 << (sizeof(int) \* 8 - 1)) - 1;        // toggle the bits in the range l to r in 'num'      num = toggleBitsFromLToR(num, l, r);        // unset the bits in the range l to r in 'n'      // and return the number      return (n & num);  }    // Driver program to test above  int main()  {      unsigned int n = 42;      unsigned int l = 2, r = 5;      cout << unsetBitsInGivenRange(n, l, r);      return 0;  } |

Run on IDE

Output:

32

# **Find the largest number with n set and m unset bits**

Given two non-negative numbers **n** and **m**. The problem is to find the largest number having **n**number of set bits and **m** number of unset bits in its binary representation.

**Note :**0 bits before leading 1 (or leftmost 1) in binary representation are counted

**Contraints:** 1 <= n, 0 <= m, (m+n) <= 31

Examples:

Input : n = 2, m = 2

Output : 12

**(12)10** = (1100)2

We can see that in the binary representation of **12**

there are 2 set and 2 unsets bits and it is the largest number.

Input : n = 4, m = 1

Output : 30

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

Following are the steps:

1. Calculate **num** = (1 << (n + m)) – 1. This will produce a number **num** having **(n + m)** number of bits and all are set.
2. Now, toggle the last **m** bits of **num** and then return the toggled number. Refer [this](http://www.geeksforgeeks.org/toggle-last-m-bits/) post.

* C/C++
* Java

|  |
| --- |
| // C++ implementation to find the largest number  // with n set and m unset bits  #include <bits/stdc++.h>    using namespace std;    // function to toggle the last m bits  unsigned int toggleLastMBits(unsigned int n,                               unsigned int m)  {      // if no bits are required to be toggled      if (m == 0)          return n;        // calculating a number 'num' having 'm' bits      // and all are set      unsigned int num = (1 << m) - 1;        // toggle the last m bits and return the number      return (n ^ num);  }    // function to find the largest number  // with n set and m unset bits  unsigned int largeNumWithNSetAndMUnsetBits(unsigned int n,                                             unsigned int m)  {      // calculating a number 'num' having '(n+m)' bits      // and all are set      unsigned int num = (1 << (n + m)) - 1;        // required largest number      return toggleLastMBits(num, m);  }    // Driver program to test above  int main()  {      unsigned int n = 2, m = 2;      cout << largeNumWithNSetAndMUnsetBits(n, m);      return 0;  } |

Run on IDE

Output:

12

# **Find the smallest number with n set and m unset bits**

Given two non-negative numbers **n** and **m**. The problem is to find the smallest number having **n**number of set bits and **m** number of unset bits in its binary representation.

**Constraints:** 1 <= n, 0 <= m, (m+n) <= 31

**Note :**0 bits before leading 1 (or leftmost 1) in binary representation are counted

Examples:

Input : n = 2, m = 2

Output : 9

**(9)10** = (1001)2

We can see that in the binary representation of **9**

there are 2 set and 2 unsets bits and it is the

smallest number.

Input : n = 4, m = 1

Output : 23

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

**Approach:** Following are the steps:

1. Calculate **num** = (1 << (n + m)) – 1. This will produce a number **num** having **(n + m)** number of bits and all are set.
2. Now, toggle bits in the range from **n** to **(n+m-1)** in **num**, i.e, to toggle bits from the rightmost **nth**bit to the rightmost **(n+m-1)th** bit and then return the toggled number. Refer [this](http://www.geeksforgeeks.org/toggle-bits-given-range/) post.

* C/C++
* Java

|  |
| --- |
| // C++ implementation to find the smallest number  // with n set and m unset bits  #include <bits/stdc++.h>    using namespace std;    // function to toggle bits in the given range  unsigned int toggleBitsFromLToR(unsigned int n,                                  unsigned int l,                                  unsigned int r)  {      // for invalid range      if (r < l)          return n;        // calculating a number 'num' having 'r'      // number of bits and bits in the range l      // to r are the only set bits      int num = ((1 << r) - 1) ^ ((1 << (l - 1)) - 1);        // toggle bits in the range l to r in 'n'      // and return the number      return (n ^ num);  }    // function to find the smallest number  // with n set and m unset bits  unsigned int smallNumWithNSetAndMUnsetBits(unsigned int n,                                             unsigned int m)  {      // calculating a number 'num' having '(n+m)' bits      // and all are set      unsigned int num = (1 << (n + m)) - 1;        // required smallest number      return toggleBitsFromLToR(num, n, n + m - 1);  }    // Driver program to test above  int main()  {      unsigned int n = 2, m = 2;      cout << smallNumWithNSetAndMUnsetBits(n, m);      return 0;  } |

Run on IDE

Output:

9

# **Check if binary representation of a given number and its complement are anagram**

Given a positive number you need to check whether it’s complement and the number are anagrams or not.

Examples:

Input : a = 4294967295

Output : Yes

Binary representation of 'a' and it's

complement are anagrams of each other

Input : a = 4

Output : No

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

**Simple Approach:**In this approach calculation of the complement of the number is allowed.

1. Find binary representation of the number and it’s complement using simple decimal to binary representation technique.

2. Sort both the binary representations and compare them to check whether they are anagrams or not.

|  |
| --- |
| // A simple C++ program to check if binary  // representations of a number and it's  // complement are anagram.  #include <bits/stdc++.h>  #define ull unsigned long long int  using namespace std;    const int ULL\_SIZE = 8\*sizeof(ull);    bool isComplementAnagram(ull a)  {      ull b = ~a; // Finding complement of a;        // Find reverse binary representation of a.      bool binary\_a[ULL\_SIZE] = { 0 };      for (int i=0; a > 0; i++)      {          binary\_a[i] = a % 2;          a /= 2;      }        // Find reverse binary representation      // of complement.      bool binary\_b[ULL\_SIZE] = { 0 };      for (int i=0; b > 0; i++)      {          binary\_b[i] = b % 2;          b /= 2;      }        // Sort binary representations and compare      // after sorting.      sort(binary\_a, binary\_a + ULL\_SIZE);      sort(binary\_b, binary\_b + ULL\_SIZE);      for (int i = 0; i < ULL\_SIZE; i++)          if (binary\_a[i] != binary\_b[i])              return false;        return true;  }    // Driver code  int main()  {      ull a = 4294967295;      cout << isComplementAnagram(a) << endl;      return 0;  } |

Run on IDE

Output:

1

**Intermediate :**

# **Swap bits in a given number**

Given a number x and two positions (from right side) in binary representation of x, write a function that swaps n bits at given two positions and returns the result. It is also given that the two sets of bits do not overlap.

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/swap-bits/0)

Let p1 and p2 be the two given positions.

Example 1  
Input:  
x = 47 (00101111)  
p1 = 1 (Start from second bit from right side)  
p2 = 5 (Start from 6th bit from right side)  
n = 3 (No of bits to be swapped)  
Output:  
227 (11100011)  
The 3 bits starting from the second bit (from right side) are  
swapped with 3 bits starting from 6th position (from right side)

Example 2  
Input:  
x = 28 (11100)  
p1 = 0 (Start from first bit from right side)  
p2 = 3 (Start from 4th bit from right side)  
n = 2 (No of bits to be swapped)  
Output:  
7 (00111)  
The 2 bits starting from 0th postion (from right side) are  
swapped with 2 bits starting from 4th position (from right side)

**Solution**  
We need to swap two sets of bits. XOR can be used in a similar way as it is used to [swap 2 numbers](http://en.wikipedia.org/wiki/XOR_swap_algorithm). Following is the algorithm.

1) Move all bits of first set to rightmost side

set1 = (x >> p1) & ((1U << n) - 1)

Here the expression (1U << n) - 1 gives a number that

contains last n bits set and other bits as 0. We do &

with this expression so that bits other than the last

n bits become 0.

2) Move all bits of second set to rightmost side

set2 = (x >> p2) & ((1U << n) - 1)

3) XOR the two sets of bits

xor = (set1 ^ set2)

4) Put the xor bits back to their original positions.

xor = (xor << p1) | (xor << p2)

5) Finally, XOR the xor with original number so

that the two sets are swapped.

result = x ^ xor

**Implementation:**

|  |
| --- |
| #include<stdio.h>    int swapBits(unsigned int x, unsigned int p1, unsigned int p2, unsigned int n)  {      /\* Move all bits of first set to rightmost side \*/      unsigned int set1 =  (x >> p1) & ((1U << n) - 1);        /\* Moce all bits of second set to rightmost side \*/      unsigned int set2 =  (x >> p2) & ((1U << n) - 1);        /\* XOR the two sets \*/      unsigned int xor = (set1 ^ set2);        /\* Put the xor bits back to their original positions \*/      xor = (xor << p1) | (xor << p2);        /\* XOR the 'xor' with the original number so that the         two sets are swapped \*/      unsigned int result = x ^ xor;        return result;  }    /\* Drier program to test above function\*/  int main()  {      int res =  swapBits(28, 0, 3, 2);      printf("\nResult = %d ", res);      return 0;  } |

Run on IDE

Output:

Result = 7

# **Add two numbers without using arithmetic operators**

Write a function Add() that returns sum of two integers. The function should not use any of the arithmetic operators (+, ++, –, -, .. etc).

Sum of two bits can be obtained by performing XOR (^) of the two bits. Carry bit can be obtained by performing AND (&) of two bits.  
Above is simple [Half Adder](http://en.wikipedia.org/wiki/Adder_%28electronics%29#Half_adder) logic that can be used to add 2 single bits. We can extend this logic for integers. If x and y don’t have set bits at same position(s), then bitwise XOR (^) of x and y gives the sum of x and y. To incorporate common set bits also, bitwise AND (&) is used. Bitwise AND of x and y gives all carry bits. We calculate (x & y) << 1 and add it to x ^ y to get the required result.

|  |
| --- |
| #include<stdio.h>    int Add(int x, int y)  {      // Iterate till there is no carry      while (y != 0)      {          // carry now contains common set bits of x and y          int carry = x & y;            // Sum of bits of x and y where at least one of the bits is not set          x = x ^ y;            // Carry is shifted by one so that adding it to x gives the required sum          y = carry << 1;      }      return x;  }    int main()  {      printf("%d", Add(15, 32));      return 0;  } |

Run on IDE

Following is recursive implementation for the same approach.

|  |
| --- |
| int Add(int x, int y)  {      if (y == 0)          return x;      else          return Add( x ^ y, (x & y) << 1);  } |

# **Smallest of three integers without comparison operators**

Write a C program to find the smallest of three integers, without using any of the comparison operators.

Let 3 input numbers be x, y and z.

**Method 1 (Repeated Subtraction)**  
Take a counter variable c and initialize it with 0. In a loop, repeatedly subtract x, y and z by 1 and increment c. The number which becomes 0 first is the smallest. After the loop terminates, c will hold the minimum of 3.

|  |
| --- |
| #include<stdio.h>    int smallest(int x, int y, int z)  {    int c = 0;    while ( x && y && z )    {        x--;  y--; z--; c++;    }    return c;  }    int main()  {     int x = 12, y = 15, z = 5;     printf("Minimum of 3 numbers is %d", smallest(x, y, z));     return 0;  } |

Run on IDE

This methid doesn’t work for negative numbers. Method 2 works for negative nnumbers also.

**Method 2 (Use Bit Operations)**  
Use method 2 of [this post to find minimum of two numbers](http://www.geeksforgeeks.org/archives/2643) (We can’t use Method 1 as Method 1 uses comparison operator). Once we have functionality to find minimum of 2 numbers, we can use this to find minimum of 3 numbers.

|  |
| --- |
| // See mthod 2 of <http://www.geeksforgeeks.org/archives/2643>  #include<stdio.h>  #define CHAR\_BIT 8    /\*Function to find minimum of x and y\*/  int min(int x, int y)  {    return  y + ((x - y) & ((x - y) >>              (sizeof(int) \* CHAR\_BIT - 1)));  }    /\* Function to find minimum of 3 numbers x, y and z\*/  int smallest(int x, int y, int z)  {      return min(x, min(y, z));  }    int main()  {     int x = 12, y = 15, z = 5;     printf("Minimum of 3 numbers is %d", smallest(x, y, z));     return 0;  } |

Run on IDE

**Method 3 (Use Division operator)**  
We can also use division operator to find minimum of two numbers. If value of (a/b) is zero, then b is greater than a, else a is greater. Thanks to [gopinath](http://www.geeksforgeeks.org/smallest-of-three-integers-without-comparison-operators/#comment-15324)and [Vignesh](http://in.linkedin.com/in/vignesh4430)for suggesting this method.

|  |
| --- |
| #include <stdio.h>    // Using division operator to find minimum of three numbers  int smallest(int x, int y, int z)  {      if (!(y/x))  // Same as "if (y < x)"          return (!(y/z))? y : z;      return (!(x/z))? x : z;  }    int main()  {      int x = 78, y = 88, z = 68;      printf("Minimum of 3 numbers is %d", smallest(x, y, z));      return 0;  } |

# **Compute the integer absolute value (abs) without branching**

We need not to do anything if a number is positive. We want to change only negative numbers. Since negative numbers are stored in [2’s complement](http://en.wikipedia.org/wiki/Two%27s_complement) form, to get the absolute value of a negative number we have to toggle bits of the number and add 1 to the result.

For example -2 in a 8 bit system is stored as follows 1 1 1 1 1 1 1 0 where leftmost bit is the sign bit. To get the absolute value of a negative number, we have to toggle all bits and add 1 to the toggled number i.e, 0 0 0 0 0 0 0 1 + 1 will give the absolute value of 1 1 1 1 1 1 1 0. Also remember, we need to do these operations only if the number is negative (sign bit is set).

**Method 1**  
1) Set the mask as right shift of integer by 31 (assuming integers are stored using 32 bits).

mask = n>>31

2) For negative numbers, above step sets mask as 1 1 1 1 1 1 1 1 and 0 0 0 0 0 0 0 0 for positive numbers. Add the mask to the given number.

mask + n

3) XOR of mask +n and mask gives the absolute value.

(mask + n)^mask

Implementation:

|  |
| --- |
| #include <stdio.h>  #define CHAR\_BIT 8    /\* This function will return absoulte value of n\*/  unsigned int getAbs(int n)  {    int const mask = n >> (sizeof(int) \* CHAR\_BIT - 1);    return ((n + mask) ^ mask);  }    /\* Driver program to test above function \*/  int main()  {    int n = -6;    printf("Absoute value of %d is %u", n, getAbs(n));      getchar();    return 0;  } |

Run on IDE

**Method 2:**  
1) Set the mask as right shift of integer by 31 (assuming integers are stored using 32 bits).

mask = n>>31

2) XOR the mask with number

mask ^ n

3) Subtract mask from result of step 2 and return the result.

(mask^n) - mask

Implementation:

|  |
| --- |
| /\* This function will return absoulte value of n\*/  unsigned int getAbs(int n)  {    int const mask = n >> (sizeof(int) \* CHAR\_BIT - 1);    return ((n ^ mask) - mask);  } |

Run on IDE

On machines where branching is expensive, the above expression can be faster than the obvious approach, r = (v < 0) ? -(unsigned)v : v, even though the number of operations is the same. Please see [this](http://graphics.stanford.edu/~seander/bithacks.html#IntegerAbs)for more details about the above two methods.

# **Write an Efficient C Program to Reverse Bits of a Number**

Given an unsigned integer, reverse all bits of it and return the number with reversed bits.

Input : n = 1

Output : 2147483648

On a machine with size of unsigned

bit as 32. Reverse of 0....001 is

100....0.

Input : n = 2147483648

Output : 1

## [Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/reverse-bits/0)

**Method1 – Simple**  
Loop through all the bits of an integer. If a bit at ith position is set in the i/p no. then set the bit at (NO\_OF\_BITS – 1) – i in o/p. Where NO\_OF\_BITS is number of bits present in the given number.

|  |
| --- |
| /\* Function to reverse bits of num \*/  unsigned int reverseBits(unsigned int num)  {      unsigned int  NO\_OF\_BITS = sizeof(num) \* 8;      unsigned int reverse\_num = 0, i, temp;        for (i = 0; i < NO\_OF\_BITS; i++)      {          temp = (num & (1 << i));          if(temp)              reverse\_num |= (1 << ((NO\_OF\_BITS - 1) - i));      }        return reverse\_num;  }    /\* Driver function to test above function \*/  int main()  {      unsigned int x = 2;      printf("%u", reverseBits(x));      getchar();  } |

Run on IDE

Above program can be optimized by removing the use of variable temp. See below the modified code.

|  |
| --- |
| unsigned int reverseBits(unsigned int num)  {      unsigned int  NO\_OF\_BITS = sizeof(num) \* 8;      unsigned int reverse\_num = 0;      int i;      for (i = 0; i < NO\_OF\_BITS; i++)      {          if((num & (1 << i)))             reverse\_num |= 1 << ((NO\_OF\_BITS - 1) - i);     }      return reverse\_num;  } |

Run on IDE

Time Complexity: O(log n)  
Space Complexity: O(1)

**Method 2 – Standard**  
The idea is to keep putting set bits of the num in reverse\_num until num becomes zero. After num becomes zero, shift the remaining bits of reverse\_num.

Let num is stored using 8 bits and num be 00000110. After the loop you will get reverse\_num as 00000011. Now you need to left shift reverse\_num 5 more times and you get the exact reverse 01100000.

|  |
| --- |
| unsigned int reverseBits(unsigned int num)  {      unsigned int count = sizeof(num) \* 8 - 1;      unsigned int reverse\_num = num;        num >>= 1;      while(num)      {         reverse\_num <<= 1;         reverse\_num |= num & 1;         num >>= 1;         count--;      }      reverse\_num <<= count;      return reverse\_num;  }    int main()  {      unsigned int x = 1;      printf("%u", reverseBits(x));      getchar();  } |

Run on IDE

Time Complexity: O(log n)  
Space Complexity: O(1)

# **Swap all odd and even bits**

Given an unsigned integer, swap all odd bits with even bits. For example, if the given number is 23 (**0**0**0**1**0**1**1**1), it should be converted to 43 (0**0**1**0**1**0**1**1**). Every even position bit is swapped with adjacent bit on right side (even position bits are highlighted in binary representation of 23), and every odd position bit is swapped with adjacent on left side.

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/swap-all-odd-and-even-bits/0)

If we take a closer look at the example, we can observe that we basically need to right shift (>>) all even bits (In the above example, even bits of 23 are highlighted) by 1 so that they become odd bits (highlighted in 43), and left shift (<<) all odd bits by 1 so that they become even bits. The following solution is based on this observation. The solution assumes that input number is stored using 32 bits.

Let the input number be x  
1) Get all even bits of x by doing bitwise and of x with 0xAAAAAAAA. The number 0xAAAAAAAA is a 32 bit number with all even bits set as 1 and all odd bits as 0.  
2) Get all odd bits of x by doing bitwise and of x with 0x55555555. The number 0x55555555 is a 32 bit number with all odd bits set as 1 and all even bits as 0.  
3) Right shift all even bits.  
4) Left shift all odd bits.  
5) Combine new even and odd bits and return.

* C/C++

|  |
| --- |
| // C program to swap even and odd bits of a given number  #include <stdio.h>    unsigned int swapBits(unsigned int x)  {      // Get all even bits of x      unsigned int even\_bits = x & 0xAAAAAAAA;        // Get all odd bits of x      unsigned int odd\_bits  = x & 0x55555555;        even\_bits >>= 1;  // Right shift even bits      odd\_bits <<= 1;   // Left shift odd bits        return (even\_bits | odd\_bits); // Combine even and odd bits  }    // Driver program to test above function  int main()  {      unsigned int x = 23; // 00010111        // Output is 43 (00101011)      printf("%u ", swapBits(x));        return 0;  } |

Run on IDE

Output:

43

# **Check if a number is multiple of 9 using bitwise operators**

Given a number n, write a function that returns true if n is divisible by 9, else false. The most simple way to check for n’s divisibility by 9 is to do n%9.   
Another method is to sum the digits of n. If sum of digits is multiple of 9, then n is multiple of 9.  
The above methods are not bitwise operators based methods and require use of % and /.  
The [bitwise operators](http://www.geeksforgeeks.org/interesting-facts-bitwise-operators-c/) are generally faster than modulo and division operators. Following is a bitwise operator based method to check divisibility by 9.

|  |
| --- |
| #include<iostream>  using namespace std;    // Bitwise operator based function to check divisibility by 9  bool isDivBy9(int n)  {      // Base cases      if (n == 0 || n == 9)          return true;      if (n < 9)          return false;        // If n is greater than 9, then recur for [floor(n/9) - n%8]      return isDivBy9((int)(n>>3) - (int)(n&7));  }    // Driver program to test above function  int main()  {      // Let us print all multiples of 9 from 0 to 100      // using above method      for (int i = 0; i < 100; i++)         if (isDivBy9(i))           cout << i << " ";      return 0;  } |

Run on IDE

Output:

0 9 18 27 36 45 54 63 72 81 90 99

**How does this work?**  
n/9 can be written in terms of n/8 using the following simple formula.

n/9 = n/8 - n/72

Since we need to use bitwise operators, we get the value of floor(n/8) using n>>3 and get value of n%8 using n&7. We need to write above expression in terms of floor(n/8) and n%8.  
n/8 is equal to “floor(n/8) + (n%8)/8”. Let us write the above expression in terms of floor(n/8) and n%8

n/9 = floor(n/8) + (n%8)/8 - [floor(n/8) + (n%8)/8]/9

n/9 = floor(n/8) - [floor(n/8) - 9(n%8)/8 + (n%8)/8]/9

n/9 = floor(n/8) - [floor(n/8) - n%8]/9

From above equation, n is a multiple of 9 only if the expression floor(n/8) – [floor(n/8) – n%8]/9 is an integer. This expression can only be an integer if the sub-expression [floor(n/8) – n%8]/9 is an integer. The subexpression can only be an integer if [floor(n/8) – n%8] is a multiple of 9. So the problem reduces to a smaller value which can be written in terms of bitwise operators.

# **Check if binary representation of a number is palindrome**

Given an integer ‘x’, write a C function that returns true if binary representation of x is palindrome else return false.

For example a numbers with binary representation as 10..01 is palindrome and number with binary representation as 10..00 is not palindrome.

The idea is similar to [checking a string is palindrome or not](http://geeksquiz.com/c-program-check-given-string-palindrome/). We start from leftmost and rightmost bits and compare bits one by one. If we find a mismatch, then return false.

## [Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/palindrome-numbers/0)

**Algorithm:**  
isPalindrome(x)  
1) Find number of bits in x using sizeof() operator.  
2) Initialize left and right positions as 1 and n respectively.  
3) Do following while left ‘l’ is smaller than right ‘r’.  
..…..a) If bit at position ‘l’ is not same as bit at position ‘r’, then return false.  
..…..b) Increment ‘l’ and decrement ‘r’, i.e., do l++ and r–-.  
4) If we reach here, it means we didn’t find a mismatching bit.

To find the bit at a given position, we can use the idea similar to [this](http://www.geeksforgeeks.org/how-to-turn-off-a-particular-bit-in-a-number/)post. The expression “**x & (1 << (k-1))**” gives us non-zero value if bit at k’th position from right is set and gives a zero value if if k’th bit is not set.

Following is C++ implementation of the above algorithm.

|  |
| --- |
| #include<iostream>  using namespace std;    // This function returns true if k'th bit in x is set (or 1).  // For example if x (0010) is 2 and k is 2, then it returns true  bool isKthBitSet(unsigned int x, unsigned int k)  {      return (x & (1 << (k-1)))? true: false;  }    // This function returns true if binary representation of x is  // palindrome. For example (1000...001) is paldindrome  bool isPalindrome(unsigned int x)  {      int l = 1; // Initialize left position      int r = sizeof(unsigned int)\*8; // initialize right position        // One by one compare bits      while (l < r)      {          if (isKthBitSet(x, l) != isKthBitSet(x, r))              return false;          l++;     r--;      }      return true;  }    // Driver program to test above function  int main()  {      unsigned int x = 1<<15 + 1<<16;      cout << isPalindrome(x) << endl;      x = 1<<31 + 1;      cout << isPalindrome(x) << endl;      return 0;  } |

Run on IDE

Output:

1

1

# **How to swap two bits in a given integer?**

Given an integer n and two bit positions p1 and p2 inside it, swap bits at the given positions. The given positions are from least significant bit (lsb). For example, the position for lsb is 0.

Examples:

Input: n = 28, p1 = 0, p2 = 3

Output: 21

28 in binary is 11100. If we swap 0'th and 3rd digits,

we get 10101 which is 21 in decimal.

Input: n = 20, p1 = 2, p2 = 3

Output: 24

**We strongly recommend you to minimize your browser and try this yourself first.**

The idea is to first find the bits, then use [XOR based swapping concept](http://www.geeksforgeeks.org/swap-two-numbers-without-using-temporary-variable/), i..e., to swap two numbers ‘x’ and ‘y’, we do x = x ^ y,  y = y ^ x and x = x ^ y.

Below is C implementation.

|  |
| --- |
| // C program to swap bits in an intger  #include<stdio.h>    // This function swaps bit at positions p1 and p2 in an integer n  int swapBits(unsigned int n, unsigned int p1, unsigned int p2)  {      /\* Move p1'th to rightmost side \*/      unsigned int bit1 =  (n >> p1) & 1;        /\* Move p2'th to rightmost side \*/      unsigned int bit2 =  (n >> p2) & 1;        /\* XOR the two bits \*/      unsigned int x = (bit1 ^ bitt2);        /\* Put the xor bit back to their original positions \*/      x = (x << p1) | (x << p2);        /\* XOR 'x' with the original number so that the         two sets are swapped \*/      unsigned int result = n ^ x;  }    /\* Drier program to test above function\*/  int main()  {      int res =  swapBits(28, 0, 3);      printf("\nResult = %d ", res);      return 0;  } |

Run on IDE

Output:

Result = 24

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

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

Examples:

Input: n = 5

Output: 25

Input: 7

Output: 49

Input: n = 12

Output: 144

A **Simple Solution** is to repeatedly add n to result. Below is C++ 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;  }    // drive program  int main()  {      for (int n = 1; n<=5; n++)          cout << "n = " << n << ", n^2 = "               << square(n) << endl;      return 0;  } |

Run on IDE

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

Time complexity of above solution is O(n). 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 bitwise right shift operator. 2\*x and 4\*x can be calculated

Below is C++ implementation based on above idea.

|  |
| --- |
| // Square of a number using bitwise operators  #include<iostream>  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);  }    // drive program  int main()  {      for (int n = 1; n<=5; n++)          cout << "n = " << n << ", n^2 = " << square(n) << endl;      return 0;  } |

Run on IDE

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

Time complexity of the above solution is O(Logn).

# **Cyclic Redundancy Check and Modulo-2 Division**

CRC or Cyclic Redundancy Check is a method of detecting accidental changes/errors in communication channel.

CRC uses **Generator Polynomial**which is available on both sender and receiver side. An example generator polynomial is of the form like x3 + x + 1. This generator polynomial represents key 1011. Another example is x2 + 1 that represents key 101.

n : Number of bits in data to be sent

from sender side.

k : Number of bits in the key obtained

from generator polynomial.

**Sender Side (Generation of Encoded Data from Data and Generator Polynomial (or Key)):**

1. The binary data is first augmented by adding k-1 zeros in the end of the data
2. Use ***modulo-2 binary division*** to divide binary data by the key and store remainder of division.
3. Append the remainder at the end of the data to form the encoded data and send the same

.

**Receiver Side (Check if there are errors introduced in transmission)**  
Perform modulo-2 division again and if remainder is 0, then there are no errors.

In this article we will focus only on finding the remainder i.e. check word and the code word.

**Modulo 2 Division:**

The process of modulo-2 binary division is the same as the familiar division process we use for decimal numbers. Just that instead of subtraction, we use XOR here.

* + - In each step, a copy of the divisor (or data) is XORed with the k bits of the dividend (or key).
    - The result of the XOR operation (remainder) is (n-1) bits, which is used for the next step after 1 extra bit is pulled down to make it n bits long.
    - When there are no bits left to pull down, we have a result. The (n-1)-bit remainder which is appended at the sender side.

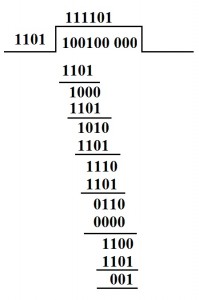
**Illustration:**

**Example 1 (No error in transmission):**

Data word to be sent - 100100

Key - 1101 [ Or generator polynomial x3 + x + 1]

Sender Side:

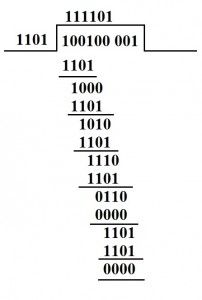
[](http://www.geeksforgeeks.org/wp-content/uploads/gq/2015/05/sender.jpg)

Therefore, the remainder is 001 and hence the encoded

data sent is 100100001.

Receiver Side:

Code word received at the receiver side 100100001

[](http://www.geeksforgeeks.org/wp-content/uploads/gq/2015/05/receiver-y.jpg)

Therefore, the remainder is all zeros. Hence, the

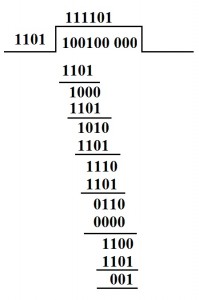
data received has no error.

**Example 2: (Error in transmission)**

Data word to be sent - 100100

Key - 1101

Sender Side:

[](http://www.geeksforgeeks.org/wp-content/uploads/gq/2015/05/sender.jpg)

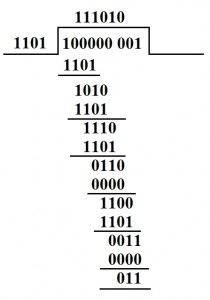
Therefore, the remainder is 001 and hence the

code word sent is 100100001.

Receiver Side

Let there be error in transmission media

Code word received at the receiver side - 100000001

[](http://www.geeksforgeeks.org/wp-content/uploads/gq/2015/05/receiver-n.jpg)

Since the remainder is not all zeroes, the error

is detected at the receiver side.

**Implementation**

Below is Python implementation for generating code word from given binary data and key.

|  |
| --- |
| # Returns XOR of 'a' and 'b'  # (both of same length)  def xor(a, b):        # initialize result      result = []        # Traverse all bits, if bits are      # same, then XOR is 0, else 1      for i in range(1, len(b)):          if a[i] == b[i]:              result.append('0')          else:              result.append('1')        return ''.join(result)      # Performs Modulo-2 division  def mod2div(divident, divisor):        # Number of bits to be XORed at a time.      pick = len(divisor)        # Slicing the divident to appropriate      # length for particular step      tmp = divident[0 : pick]        while pick < len(divident):            if tmp[0] == '1':                # replace the divident by the result              # of XOR and pull 1 bit down              tmp = xor(divisor, tmp) + divident[pick]            else:   # If leftmost bit is '0'              # If the leftmost bit of the dividend (or the              # part used in each step) is 0, the step cannot              # use the regular divisor; we need to use an              # all-0s divisor.              tmp = xor('0'\*pick, tmp) + divident[pick]            # increment pick to move further          pick += 1        # For the last n bits, we have to carry it out      # normally as increased value of pick will cause      # Index Out of Bounds.      if tmp[0] == '1':          tmp = xor(divisor, tmp)      else:          tmp = xor('0'\*pick, tmp)        checkword = tmp      return checkword    # Function used at the sender side to encode  # data by appending remainder of modular divison  # at the end of data.  def encodeData(data, key):        l\_key = len(key)        # Appends n-1 zeroes at end of data      appended\_data = data + '0'\*(l\_key-1)      remainder = mod2div(appended\_data, key)        # Append remainder in the original data      codeword = data + remainder      print("Remainder : ", remainder)      print("Encoded Data (Data + Remainder) : ",            codeword)    # Driver code  data = "100100"  key = "1101"  encodeData(data, key) |

Run on IDE

Output:

Remainder : 001

Encoded Data (Data + Remainder) : 100100001

Note that CRC is mainly designed and used to protect against common of errors on communication channels and NOT suitable protection against intentional alteration of data (See reasons [here](https://en.wikipedia.org/wiki/Cyclic_redundancy_check#Data_integrity))

# **Copy set bits in a range**

Given two numbers x and y, and a range [l, r] where 1 <= l, r <= 32. The task is consider set bits of y in range [l, r] and set these bits in x also. Examples :

Input : x = 10, y = 13, l = 2, r = 3

Output : x = 14

Binary representation of 10 is 1**01**0 and

that of y is 1**10**1. There is one set bit

in y at 3’rd position (in given range).

After we copy this bit to x, x becomes 1**11**0

which is binary representation of 14.

Input : x = 8, y = 7, l = 1, r = 2

Output : x = 11

Source : [D E Shaw Interview](http://www.geeksforgeeks.org/d-e-shaw-interview-experience-set-16-on-campus-for-internship/)

## [Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/copy-set-bits-in-range/0)

**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 ;       // Travers 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 = 2, r = 3;     copySetBits(x, y, l, r);     cout << "Modified x is " << x;     return 0;  } |

Run on IDE

Output :

Modified x is 14

**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 = (1<<(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;  } |

Run on IDE

Output :

Modified x is 14

# **Check if a number is Bleak**

A number ‘n’ is called Bleak if it cannot be represented as sum of a positive number x and set bit count in x, i.e., x + [countSetBits(x)](http://www.geeksforgeeks.org/count-set-bits-in-an-integer/) is not equal to n for any non-negative number x.

Examples :

Input : n = 3

Output : false

3 is not Bleak as it can be represented

as 2 + countSetBits(2).

Input : n = 4

Output : true

4 is t Bleak as it cannot be represented

as sum of a number x and countSetBits(x)

for any number x.

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/bleak-numbers/0)

**Method 1 (Simple)**

bool isBleak(n)

1) Consider all numbers smaller than n

a) If x + [countSetBits(x)](http://www.geeksforgeeks.org/count-set-bits-in-an-integer/) == n

return false

2) Return true

Below is C++ implementation of the simple approach.

|  |
| --- |
| // A simple C++ program to check Bleak Number  #include <bits/stdc++.h>  using namespace std;    /\* Function to get no of set bits in binary     representation of passed binary no. \*/  int countSetBits(int x)  {      unsigned int count = 0;      while (x)      {        x &= (x-1) ;        count++;      }      return count;  }    // Returns true if n is Bleak  bool isBleak(int n)  {     // Check for all numbers 'x' smaller     // than n.  If x + countSetBits(x)     // becomes n, then n can't be Bleak     for (int x=1; x<n; x++)        if (x + countSetBits(x) == n)            return false;       return true;  }    // Driver code  int main()  {    isBleak(3)? cout << "Yes\n" : cout << "No\n";    isBleak(4)? cout << "Yes\n" : cout << "No\n";    return 0;  } |

Run on IDE

Output :

No

Yes

# **Count trailing zero bits using lookup table**

Given an integer, count the number of trailing zeroes. For example, for n = 12, its binary representation is 1100 and number of trailing zero bits is 2.

Examples:

Input : 8

Output : 3

Binary of 8 is 1000, so there are theree

trailing zero bits.

Input : 18

Output : 1

Binary of 10 is 10010, so there is one

trailing zero bit.

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

A **simple solution**is to traverse bits from LSB (Least Significant Bit) and increment count while bit is 0.

|  |
| --- |
| // Simple C++ code for counting trailing zeros  // in binary representation of a number  #include<bits/stdc++.h>  using namespace std;    int countTrailingZero(int x)  {    int count = 0;    while ((x & 1) == 0)    {        x = x >> 1;        count++;    }    return count;  }    // Driver Code  int main()  {      cout << countTrailingZero(11) << endl;      return 0;  } |

Run on IDE

Output :

4

Time Complexity : O(Log n)

The **lookup table solution** is based on following concepts :

1. The solution assumes that negative numbers are stored in [2’s complement](http://www.geeksforgeeks.org/efficient-method-2s-complement-binary-string/) form which is true for most of the devices. If numbers are represented in 2’s complement form, then (x & -x) [Bitwise and of x and minus x] produces a number with only last set bit.
2. Once we get a number with only one bit set, we can find its position using lookup table. It makes use of the fact that the first 32 bit position values are relatively prime with 37, so performing a modulus division with 37 gives a unique number from 0 to 36 for each. These numbers may then be mapped to the number of zeros using a small lookup table.

|  |
| --- |
| // C++ code for counting trailing zeros  // in binary representation of a number  #include<bits/stdc++.h>  using namespace std;    int countTrailingZero(int x)  {       // Map a bit value mod 37 to its position       static const int lookup[] = {32, 0, 1,       26, 2, 23, 27, 0, 3, 16, 24, 30, 28, 11,       0, 13, 4, 7, 17, 0, 25, 22, 31, 15, 29,       10, 12, 6, 0, 21, 14, 9, 5, 20, 8, 19,       18};         // Only difference between (x and -x) is       // the value of signed magnitude(leftmostbit)       // negative numbers signed bit is 1       return lookup[(-x & x) % 37];  }    // Driver Code  int main()  {      cout << countTrailingZero(48) << endl;      return 0;  } |

Run on IDE

Output :

4

Time Complexity = O(1)

# **Multiplication of two numbers with shift operator**

For any given two numbers n and m, you have to find n\*m without using any multiplication operator.

Examples:

Input: n = 25 , m = 13

Output: 325

Input: n = 50 , m = 16

Output: 800

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

We can solve this problem with the shift operator. The idea is based on the fact that every number can be represented in binary form. And multiplication with a number is equivalent to multiplication with powers of 2. Powers of 2 can be obtained using left shift operator.

Check for every set bit in the binary representation of m and for every set bit left shift n, count times where count if place value of the set bit of m and add that value to answer.

|  |
| --- |
| // CPP program to find multiplication  // of two number without use of  // multiplication operator  #include<bits/stdc++.h>  using namespace std;    // Function for multiplication  int multiply(int n, int m)  {      int ans = 0, count = 0;      while (m)      {          // check for set bit and left          // shift n, count times          if (m % 2 == 1)              ans += n << count;            // increment of place value (count)          count++;          m /= 2;      }      return ans;  }    // Driver program  int main()  {      int n = 20 , m = 13;      cout << multiply(n, m);      return 0;  } |

Run on IDE

Output:

260

Time Complexity : O(log n)

**Hard :**

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

Given a positive integer n, count the total number of set bits in binary representation of all numbers from 1 to n.

Examples:

Input: n = 3

Output: 4

Input: n = 6

Output: 9

Input: n = 7

Output: 12

Input: n = 8

Output: 13

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/count-total-set-bits/0)

Source: Amazon Interview Question

**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 <stdio.h>    // 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;     printf ("Total set bit count is %d", countSetBits(n));     return 0;  } |

Run on IDE

Output:

Total set bit count is 5

Time Complexity: O(nLogn)

**Method 2 (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.

|  |
| --- |
| // A O(Logn) complexity program to count set bits in all numbers from 1 to n  #include <stdio.h>    /\* 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 program to test above functions  int main()  {     int n = 17;     printf ("Total set bit count is %d", countSetBits(n));     return 0;  } |

Run on IDE

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.

# **Program to count number of set bits in an (big) array**

Given an integer array of length N (an arbitrarily large number). How to count number of set bits in the array?

The simple approach would be, create an efficient method to count set bits in a word (most prominent size, usually equal to bit length of processor), and add bits from individual elements of array.

Various methods of counting set bits of an integer exists, see [this](http://www.geeksforgeeks.org/archives/1176) for example. These methods run at best O(logN) where N is number of bits. Note that on a processor N is fixed, count can be done in O(1) time on 32 bit machine irrespective of total set bits. Overall, the bits in array can be computed in O(n) time, where ‘n’ is array size.

However, a table look up will be more efficient method when array size is large. Storing table look up that can handle 232 integers will be impractical.

The following code illustrates simple program to count set bits in a randomly generated 64 K integer array. The idea is to generate a look up for first 256 numbers (one byte), and break every element of array at byte boundary. A meta program using C/C++ preprocessor generates the look up table for counting set bits in a byte.

The mathematical derivation behind meta program is evident from the following table (Add the column and row indices to get the number, then look into the table to get set bits in that number. For example, to get set bits in 10, it can be extracted from row named as 8 and column named as 2),

   0, 1, 2, 3

0 - 0, 1, 1, 2 -------- GROUP\_A(0)

4 - 1, 2, 2, 3 -------- GROUP\_A(1)

8 - 1, 2, 2, 3 -------- GROUP\_A(1)

12 - 2, 3, 3, 4 -------- GROUP\_A(2)

16 - 1, 2, 2, 3 -------- GROUP\_A(1)

20 - 2, 3, 3, 4 -------- GROUP\_A(2)

24 - 2, 3, 3, 4 -------- GROUP\_A(2)

28 - 3, 4, 4, 5 -------- GROUP\_A(3) ... so on

From the table, there is a patten emerging in multiples of 4, both in the table as well as in the group parameter. The sequence can be generalized as shown in the code.

**Complexity:**

All the operations takes O(1) except iterating over the array. The time complexity is O(n) where ‘n’ is size of array. Space complexity depends on the meta program that generates look up.

**Code:**

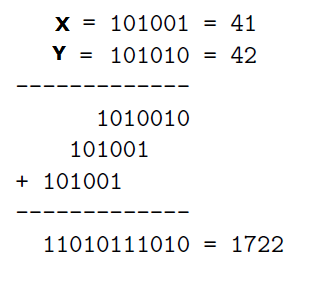
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>    /\* Size of array 64 K \*/  #define SIZE (1 << 16)    /\* Meta program that generates set bit count     array of first 256 integers \*/    /\* GROUP\_A - When combined with META\_LOOK\_UP     generates count for 4x4 elements \*/    #define GROUP\_A(x) x, x + 1, x + 1, x + 2    /\* GROUP\_B - When combined with META\_LOOK\_UP     generates count for 4x4x4 elements \*/    #define GROUP\_B(x) GROUP\_A(x), GROUP\_A(x+1), GROUP\_A(x+1), GROUP\_A(x+2)    /\* GROUP\_C - When combined with META\_LOOK\_UP     generates count for 4x4x4x4 elements \*/    #define GROUP\_C(x) GROUP\_B(x), GROUP\_B(x+1), GROUP\_B(x+1), GROUP\_B(x+2)    /\* Provide appropriate letter to generate the table \*/    #define META\_LOOK\_UP(PARAMETER) \     GROUP\_##PARAMETER(0),  \     GROUP\_##PARAMETER(1),  \     GROUP\_##PARAMETER(1),  \     GROUP\_##PARAMETER(2)   \    int countSetBits(int array[], size\_t array\_size)  {     int count = 0;       /\* META\_LOOK\_UP(C) - generates a table of 256 integers whose        sequence will be number of bits in i-th position        where 0 <= i < 256     \*/        /\* A static table will be much faster to access \*/         static unsigned char const look\_up[] = { META\_LOOK\_UP(C) };        /\* No shifting funda (for better readability) \*/      unsigned char \*pData = NULL;       for(size\_t index = 0; index < array\_size; index++)     {        /\* It is fine, bypass the type system \*/        pData = (unsigned char \*)&array[index];          /\* Count set bits in individual bytes \*/        count += look\_up[pData[0]];        count += look\_up[pData[1]];        count += look\_up[pData[2]];        count += look\_up[pData[3]];     }       return count;  }    /\* Driver program, generates table of random 64 K numbers \*/  int main()  {     int index;     int random[SIZE];       /\* Seed to the random-number generator \*/     srand((unsigned)time(0));       /\* Generate random numbers. \*/     for( index = 0; index < SIZE; index++ )     {        random[index] = rand();     }       printf("Total number of bits = %d\n", countSetBits(random, SIZE));     return 0;  } |

# **Divide and Conquer | Set 4 (Karatsuba algorithm for fast multiplication)**

Given two binary strings that represent value of two integers, find the product of two strings. For example, if the first bit string is “1100” and second bit string is “1010”, output should be 120.

For simplicity, let the length of two strings be same and be n.

A **Naive Approach** is to follow the process we study in school. One by one take all bits of second number and multiply it with all bits of first number. Finally add all multiplications. This algorithm takes O(n^2) time.

[](http://www.geeksforgeeks.org/wp-content/uploads/product.png)

Using **Divide and Conquer**, we can multiply two integers in less time complexity. We divide the given numbers in two halves. Let the given numbers be X and Y.

For simplicity let us assume that n is even

X = Xl\*2n/2 + Xr [Xl and Xr contain leftmost and rightmost n/2 bits of X]

Y = Yl\*2n/2 + Yr [Yl and Yr contain leftmost and rightmost n/2 bits of Y]

The product XY can be written as following.

XY = (Xl\*2n/2 + Xr)(Yl\*2n/2 + Yr)

= 2n XlYl + 2n/2(XlYr + XrYl) + XrYr

If we take a look at the above formula, there are four multiplications of size n/2, so we basically divided the problem of size n into for sub-problems of size n/2. But that doesn’t help because solution of recurrence T(n) = 4T(n/2) + O(n) is O(n^2). The tricky part of this algorithm is to change the middle two terms to some other form so that only one extra multiplication would be sufficient. The following is tricky expression for middle two terms.

XlYr + XrYl = (Xl + Xr)(Yl + Yr) - XlYl- XrYr

So the final value of XY becomes

XY = 2n XlYl + 2n/2 \* [(Xl + Xr)(Yl + Yr) - XlYl - XrYr] + XrYr

With above trick, the recurrence becomes T(n) = 3T(n/2) + O(n) and solution of this recurrence is O(n1.59).

What if the lengths of input strings are different and are not even? To handle the different length case, we append 0’s in the beginning. To handle odd length, we put floor(n/2) bits in left half and ceil(n/2) bits in right half. So the expression for XY changes to following.

XY = 22ceil(n/2) XlYl + 2ceil(n/2) \* [(Xl + Xr)(Yl + Yr) - XlYl - XrYr] + XrYr

The above algorithm is called Karatsuba algorithm and it can be used for any base.

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/karatsuba-algorithm/0)

Following is C++ implementation of above algorithm.

|  |
| --- |
| // C++ implementation of Karatsuba algorithm for bit string multiplication.  #include<iostream>  #include<stdio.h>    using namespace std;    // FOLLOWING TWO FUNCTIONS ARE COPIED FROM <http://goo.gl/q0OhZ>  // Helper method: given two unequal sized bit strings, converts them to  // same length by adding leading 0s in the smaller string. Returns the  // the new length  int makeEqualLength(string &str1, string &str2)  {      int len1 = str1.size();      int len2 = str2.size();      if (len1 < len2)      {          for (int i = 0 ; i < len2 - len1 ; i++)              str1 = '0' + str1;          return len2;      }      else if (len1 > len2)      {          for (int i = 0 ; i < len1 - len2 ; i++)              str2 = '0' + str2;      }      return len1; // If len1 >= len2  }    // The main function that adds two bit sequences and returns the addition  string addBitStrings( string first, string second )  {      string result;  // To store the sum bits        // make the lengths same before adding      int length = makeEqualLength(first, second);      int carry = 0;  // Initialize carry        // Add all bits one by one      for (int i = length-1 ; i >= 0 ; i--)      {          int firstBit = first.at(i) - '0';          int secondBit = second.at(i) - '0';            // boolean expression for sum of 3 bits          int sum = (firstBit ^ secondBit ^ carry)+'0';            result = (char)sum + result;            // boolean expression for 3-bit addition          carry = (firstBit&secondBit) | (secondBit&carry) | (firstBit&carry);      }        // if overflow, then add a leading 1      if (carry)  result = '1' + result;        return result;  }    // A utility function to multiply single bits of strings a and b  int multiplyiSingleBit(string a, string b)  {  return (a[0] - '0')\*(b[0] - '0');  }    // The main function that multiplies two bit strings X and Y and returns  // result as long integer  long int multiply(string X, string Y)  {      // Find the maximum of lengths of x and Y and make length      // of smaller string same as that of larger string      int n = makeEqualLength(X, Y);        // Base cases      if (n == 0) return 0;      if (n == 1) return multiplyiSingleBit(X, Y);        int fh = n/2;   // First half of string, floor(n/2)      int sh = (n-fh); // Second half of string, ceil(n/2)        // Find the first half and second half of first string.      // Refer <http://goo.gl/lLmgn> for substr method      string Xl = X.substr(0, fh);      string Xr = X.substr(fh, sh);        // Find the first half and second half of second string      string Yl = Y.substr(0, fh);      string Yr = Y.substr(fh, sh);        // Recursively calculate the three products of inputs of size n/2      long int P1 = multiply(Xl, Yl);      long int P2 = multiply(Xr, Yr);      long int P3 = multiply(addBitStrings(Xl, Xr), addBitStrings(Yl, Yr));        // Combine the three products to get the final result.      return P1\*(1<<(2\*sh)) + (P3 - P1 - P2)\*(1<<sh) + P2;  }    // Driver program to test aboev functions  int main()  {      printf ("%ld\n", multiply("1100", "1010"));      printf ("%ld\n", multiply("110", "1010"));      printf ("%ld\n", multiply("11", "1010"));      printf ("%ld\n", multiply("1", "1010"));      printf ("%ld\n", multiply("0", "1010"));      printf ("%ld\n", multiply("111", "111"));      printf ("%ld\n", multiply("11", "11"));  } |

Run on IDE

Output:

120

60

30

10

0

49

9

**Time Complexity:** Time complexity of the above solution is O(n1.59).

# **Find the maximum subarray XOR in a given array**

Given an array of integers. find the maximum XOR subarray value in given array. Expected time complexity O(n).

Examples:

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

Output: 7

The subarray {3, 4} has maximum XOR value

Input: arr[] = {8, 1, 2, 12, 7, 6}

Output: 15

The subarray {1, 2, 12} has maximum XOR value

Input: arr[] = {4, 6}

Output: 6

The subarray {6} has maximum XOR value

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/rotate-bits/0)

A **Simple Solution** is to use two loops to find XOR of all subarrays and return the maximum.

* C++
* Java

|  |
| --- |
| // A simple C++ program to find max subarray XOR  #include<bits/stdc++.h>  using namespace std;    int maxSubarrayXOR(int arr[], int n)  {      int ans = INT\_MIN;     // Initialize result        // Pick starting points of subarrays      for (int i=0; i<n; i++)      {          int curr\_xor = 0; // to store xor of current subarray            // Pick ending points of subarrays starting with i          for (int j=i; j<n; j++)          {              curr\_xor = curr\_xor ^ arr[j];              ans = max(ans, curr\_xor);          }      }      return ans;  }    // Driver program to test above functions  int main()  {      int arr[] = {8, 1, 2, 12};      int n = sizeof(arr)/sizeof(arr[0]);      cout << "Max subarray XOR is " << maxSubarrayXOR(arr, n);      return 0;  } |

Run on IDE

Output:

Max subarray XOR is 15

Time Complexity of above solution is O(n2).

An **Efficient Solution** can solve the above problem in O(n) time under the assumption that integers take fixed number of bits to store. The idea is to use Trie Data Structure. Below is algorithm.

1) Create an empty Trie. Every node of Trie is going to

contain two children, for 0 and 1 value of bit.

2) Initialize pre\_xor = 0 and insert into the Trie.

3) Initialize result = minus infinite

4) Traverse the given array and do following for every

array element arr[i].

a) pre\_xor = pre\_xor ^ arr[i]

pre\_xor now contains xor of elements from

arr[0] to arr[i].

b) Query the maximum xor value ending with arr[i]

from Trie.

c) Update result if the value obtained in step

4.b is more than current value of result.

**How does 4.b work?**  
We can observe from above algorithm that we build a Trie that contains XOR of all prefixes of given array. To find the maximum XOR subarray ending with arr[i], there may be two cases.  
i) The prefix itself has the maximum XOR value ending with arr[i]. For example if i=2 in {8, 2, 1, 12}, then the maximum subarray xor ending with arr[2] is the whole prefix.  
ii) We need to remove some prefix (ending at index from 0 to i-1). For example if i=3 in {8, 2, 1, 12}, then the maximum subarray xor ending with arr[3] starts with arr[1] and we need to remove arr[0].

To find the prefix to be removed, we find the entry in Trie that has maximum XOR value with current prefix. If we do XOR of such previous prefix with current prefix, we get the maximum XOR value ending with arr[i].  
If there is no prefix to be removed (case i), then we return 0 (that’s why we inserted 0 in Trie).

Below is C++ implementation of above algorithm.

* C++
* "Java"

|  |
| --- |
| // C++ program for a Trie based O(n) solution to find max  // subarray XOR  #include<bits/stdc++.h>  using namespace std;    // Assumed int size  #define INT\_SIZE 32    // A Trie Node  struct TrieNode  {      int value;  // Only used in leaf nodes      TrieNode \*arr[2];  };    // Utility function tp create a Trie node  TrieNode \*newNode()  {      TrieNode \*temp = new TrieNode;      temp->value = 0;      temp->arr[0] = temp->arr[1] = NULL;      return temp;  }    // Inserts pre\_xor to trie with given root  void insert(TrieNode \*root, int pre\_xor)  {      TrieNode \*temp = root;        // Start from the msb, insert all bits of      // pre\_xor into Trie      for (int i=INT\_SIZE-1; i>=0; i--)      {          // Find current bit in given prefix          bool val = pre\_xor & (1<<i);            // Create a new node if needed          if (temp->arr[val] == NULL)              temp->arr[val] = newNode();            temp = temp->arr[val];      }        // Store value at leaf node      temp->value = pre\_xor;  }    // Finds the maximum XOR ending with last number in  // prefix XOR 'pre\_xor' and returns the XOR of this maximum  // with pre\_xor which is maximum XOR ending with last element  // of pre\_xor.  int query(TrieNode \*root, int pre\_xor)  {      TrieNode \*temp = root;      for (int i=INT\_SIZE-1; i>=0; i--)      {          // Find current bit in given prefix          bool val = pre\_xor & (1<<i);            // Traverse Trie, first look for a          // prefix that has opposite bit          if (temp->arr[1-val]!=NULL)              temp = temp->arr[1-val];            // If there is no prefix with opposite          // bit, then look for same bit.          else if (temp->arr[val] != NULL)              temp = temp->arr[val];      }      return pre\_xor^(temp->value);  }    // Returns maximum XOR value of a subarray in arr[0..n-1]  int maxSubarrayXOR(int arr[], int n)  {      // Create a Trie and insert 0 into it      TrieNode \*root = newNode();      insert(root, 0);        // Initialize answer and xor of current prefix      int result = INT\_MIN, pre\_xor =0;        // Traverse all input array element      for (int i=0; i<n; i++)      {          // update current prefix xor and insert it into Trie          pre\_xor = pre\_xor^arr[i];          insert(root, pre\_xor);            // Query for current prefix xor in Trie and update          // result if required          result = max(result, query(root, pre\_xor));      }      return result;  }    // Driver program to test above functions  int main()  {      int arr[] = {8, 1, 2, 12};      int n = sizeof(arr)/sizeof(arr[0]);      cout << "Max subarray XOR is " << maxSubarrayXOR(arr, n);      return 0;  } |

Run on IDE

Output:

Max subarray XOR is 15

# **Inserting m into n such that m starts at bit j and ends at bit i.**

We are given two numbers n and m, and two-bit positions, i and j. Insert bits of m into n starting from j to i. We can assume that the bits j through i have enough space to fit all of m. That is, if m = 10011, you can assume that there are at least 5 bits between j and i. You would not, for example, have j = 3 and i = 2, because m could not fully fit between bit 3 and bit 2.

Examples:

Input : n = 1024

m = 19

i = 2

j = 6;

Output : n = 1100

Binary representations of input numbers

m in binary is (10011)2

n in binary is (10000000000)2

Binary representations of output number

(10000000000)2

Input : n = 5

m = 3

i = 1

j = 2

Output : 7

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

**Algorithm :**

1. Clear the bits j through i in n

2. Shift m so that it lines up with bits j through i

3. Return Bitwise AND of m and n.

The trickiest part is Step 1. How do we clear the bits in n? We can do this with a mask. This mask will have all 1s, except for 0s in the bits j through i. We create this mask by creating the left half of the mask first, and then the right half.

Following is C++ implementation of the above approach.

* C++
* Java

|  |
| --- |
| // C++ program for implementation of updateBits()  #include <bits/stdc++.h>  using namespace std;    // Function to updateBits M insert to N.  int updateBits(int n, int m, int i, int j)  {      /\* Create a mask to clear bits i through j        in n. EXAMPLE: i = 2, j = 4. Result        should be 11100011. For simplicity, we'll        use just 8 bits for the example. \*/        int allOnes = ~0; // will equal sequence of all ls        // ls before position j, then 0s. left = 11100000      int left= allOnes << (j + 1);        // l's after position i. right = 00000011      int right = ((1 << i) - 1);        // All ls, except for 0s between i and j. mask 11100011      int mask = left | right;        /\* Clear bits j through i then put min there \*/      int n\_cleared = n & mask; // Clear bits j through i.      int m\_shifted = m << i;   // Move m into correct position.        return (n\_cleared | m\_shifted); // OR them, and we're done!  }    // Driver Code  int main()  {      int n = 1024; // in Binary N= 10000000000      int m = 19;   // in Binary M= 10011      int i = 2, j = 6;        cout << updateBits(n,m,i,j);        return 0;  } |

Run on IDE

Output:

1100 // in Binary (10001001100)2

# **Interesting Facts about Bitwise Operators in C**

In C, following 6 operators are bitwise operators (work at bit-level)

**& (bitwise AND)** Takes two numbers as operand and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

**| (bitwise OR)** Takes two numbers as operand and does OR on every bit of two numbers. The result of OR is 1 any of the two bits is 1.

**^ (bitwise XOR)** Takes two numbers as operand and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

**<< (left shift)** Takes two numbers, left shifts the bits of first operand, the second operand decides the number of places to shift.

**>> (right shift)** Takes two numbers, right shifts the bits of first operand, the second operand decides the number of places to shift.

**~ (bitwise NOT)** Takes one number and inverts all bits of it

Following is example C program.

|  |
| --- |
| /\* C Program to demonstrate use of bitwise operators \*/  #include<stdio.h>  int main()  {      unsigned char a = 5, b = 9; // a = 4(00000101), b = 8(00001001)      printf("a = %d, b = %d\n", a, b);      printf("a&b = %d\n", a&b); // The result is 00000001      printf("a|b = %d\n", a|b);  // The result is 00001101      printf("a^b = %d\n", a^b); // The result is 00001100      printf("~a = %d\n", a = ~a);   // The result is 11111010      printf("b<<1 = %d\n", b<<1);  // The result is 00010010      printf("b>>1 = %d\n", b>>1);  // The result is 00000100      return 0;  } |

Run on IDE

Output:

a = 5, b = 9

a&b = 1

a|b = 13

a^b = 12

~a = 250

b<<1 = 18

b>>1 = 4

Following are interesting facts about bitwise operators.

**1) The left shift and right shift operators should not be used for negative numbers** The result of << and >> is undefined behabiour if any of the operands is a negative number. For example results of both -1 << 1 and 1 << -1 is undefined. Also, if the number is shifted more than the size of integer, the behaviour is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits. See [this](https://www.securecoding.cert.org/confluence/display/seccode/INT34-C.+Do+not+shift+a+negative+number+of+bits+or+more+bits+than+exist+in+the+operand) for more details.

**2) The bitwise XOR operator is the most useful operator from technical interview perspective.**It is used in many problems. A simple example could be “Given a set of numbers where all elements occur even number of times except one number, find the odd occuring number” This problem can be efficiently solved by just doing XOR of all numbers.

|  |
| --- |
| // Function to return the only odd occurring element  int findOdd(int arr[], int n) {     int res = 0, i;     for (i = 0; i < n; i++)       res ^= arr[i];     return res;  }    int main(void) {     int arr[] = {12, 12, 14, 90, 14, 14, 14};     int n = sizeof(arr)/sizeof(arr[0]);     printf ("The odd occurring element is %d ", findOdd(arr, n));     return 0;  }  // Output: The odd occurring element is 90 |

Run on IDE

The following are many other interesting problems which can be used using XOR operator.  
[Find the Missing Number](http://www.geeksforgeeks.org/find-the-missing-number/), [swap two numbers without using a temporary variable](http://www.geeksforgeeks.org/swap-two-numbers-without-using-temporary-variable/), [A Memory Efficient Doubly Linked List](http://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/), and [Find the two non-repeating elements](http://www.geeksforgeeks.org/find-two-non-repeating-elements-in-an-array-of-repeating-elements/). There are many more (See [this](http://www.geeksforgeeks.org/find-the-two-numbers-with-odd-occurences-in-an-unsorted-array/), [this](http://www.geeksforgeeks.org/add-two-numbers-without-using-arithmetic-operators/), [this](http://www.geeksforgeeks.org/swap-bits-in-a-given-number/), [this](http://www.geeksforgeeks.org/count-number-of-bits-to-be-flipped-to-convert-a-to-b/), [this](http://www.geeksforgeeks.org/find-the-element-that-appears-once/) and [this](http://www.geeksforgeeks.org/detect-if-two-integers-have-opposite-signs/))

**3) The bitwise operators should not be used in-place of logical operators.**  
The result of logical operators (&&, || and !) is either 0 or 1, but bitwise operators return an integer value. Also, the logical operators consider any non-zero operand as 1. For example consider the following program, the results of & and && are different for same operands.

|  |
| --- |
| int main()  {     int x = 2, y = 5;     (x & y)? printf("True ") : printf("False ");     (x && y)? printf("True ") : printf("False ");     return 0;  }  // Output: False True |

Run on IDE

**4) The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively.**  
As mentioned in point 1, it works only if numbers are positive.

|  |
| --- |
| int main()  {     int x = 19;     printf ("x << 1 = %d\n", x << 1);     printf ("x >> 1 = %d\n", x >> 1);     return 0;  }  // Output: 38 9 |

Run on IDE

**5) The & operator can be used to quickly check if a number is odd or even**  
The value of expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.

|  |
| --- |
| int main()  {     int x = 19;     (x & 1)? printf("Odd"): printf("Even");     return 0;  }  // Output: Odd |

Run on IDE

**6) The ~ operator should be used carefully**  
The result of ~ operator on a small number can be a big number if result is stored in a unsigned variable. And result may be negative number if result is stored in signed variable (assuming that the negative numbers are stored in 2’s complement form where leftmost bit is the sign bit)

|  |
| --- |
| // Note that the output of following program is compiler dependent  int main()  {     unsigned int x = 1;     printf("Signed Result %d \n", ~x);     printf("Unsigned Result %ud \n", ~x);     return 0;  }  /\* Output:  Signed Result -2  Unsigned Result 4294967294d \*/ |

# **What are the differences between bitwise and logical AND operators in C/C++?**

A Bitwise And operator is represented as ‘&’ and a logical operator is represented as ‘&&’. Following are some basic differences between the two operators.

**a)**The logical and operator ‘&&’ expects its operands to be boolean expressions (either 1 or 0) and returns a boolean value.  
The bitwise and operator ‘&’ works on Integral (short, int, unsigned, char, bool, unsigned char, long) values and return Integral value.

|  |
| --- |
| int main()  {      int x = 3;  //...0011      int y = 7;  //...0111        // A typical use of '&&'      if (y > 1 && y > x)        printf("y is greater than 1 AND y\n");        // A typical use of '&'      int z = x & y;   // 0011        printf ("z = %d", z);        return 0;  } |

Run on IDE

Output

y is greater than 1 AND y

z = 3.

**b)** If an integral value is used as an operand for ‘&&’ which is supposed to work on boolean values, following rule is used in C.  
…..A zero is considered as false and non-zero is considered as true.  
For example in the following program x and y are considered as 1.

|  |
| --- |
| // Example that uses non-boolean expression as  // operand for '&&'  int main()  {     int x = 2, y = 5;     printf("%d", x&&y);     return 0;  } |

Run on IDE

Output

1

It is compiler error to use non-integral expression as operand for bitwise &. For example the following program shows compiler error.

|  |
| --- |
| // Example that uses non-integral expression as  // operator for '&'  int main()  {     float x = 2.0, y = 5.0;     printf("%d", x&y);     return 0;  } |

Run on IDE

Output:

error: invalid operands to binary & (have 'float' and 'float')

**c)** The ‘&&’ operator doesn’t evaluate second operand if first operand becomes false. Similarly ‘||’ doesn’t evaluate second operand when first operand becomes true. The bitwise ‘&’ and ‘|’ operators always evaluate their operands.

|  |
| --- |
| int main()  {     int x = 0;       // 'Geeks in &&' is NOT printed because x is 0     printf("%d\n", (x && printf("Geeks in && ")) );       // 'Geeks in &' is  printed     printf("%d\n", (x  & printf("Geeks in & ")) );       return 0;  } |

Run on IDE

Output:

0

Geeks in & 0

# **Bit Fields in C**

In C, we can specify size (in bits) of structure and union members. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is withing a small range.

For example, consider the following declaration of date without use of bit fields.

|  |
| --- |
| #include <stdio.h>    // A simple representation of date  struct date  {     unsigned int d;     unsigned int m;     unsigned int y;  };    int main()  {     printf("Size of date is %d bytes\n", sizeof(struct date));     struct date dt = {31, 12, 2014};     printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);  } |

Run on IDE

Output:

Size of date is 12 bytes

Date is 31/12/2014

The above representation of ‘date’ takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, value of m is from 1 to 12, we can optimize the space using bit fields.

|  |
| --- |
| #include <stdio.h>    // A space optimized representation of date  struct date  {     // d has value between 1 and 31, so 5 bits     // are sufficient     unsigned int d: 5;       // m has value between 1 and 12, so 4 bits     // are sufficient     unsigned int m: 4;       unsigned int y;  };    int main()  {     printf("Size of date is %d bytes\n", sizeof(struct date));     struct date dt = {31, 12, 2014};     printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);     return 0;  } |

Run on IDE

Output:

Size of date is 8 bytes

Date is 31/12/2014

**Following are some interesting facts about bit fields in C.**

**1)** A special unnamed bit field of size 0 is used to force alignment on next boundary. For example consider the following program.

|  |
| --- |
| #include <stdio.h>    // A structure without forced alignment  struct test1  {     unsigned int x: 5;     unsigned int y: 8;  };    // A structure with forced alignment  struct test2  {     unsigned int x: 5;     unsigned int: 0;     unsigned int y: 8;  };    int main()  {     printf("Size of test1 is %d bytes\n", sizeof(struct test1));     printf("Size of test2 is %d bytes\n", sizeof(struct test2));     return 0;  } |

Run on IDE

Output:

Size of test1 is 4 bytes

Size of test2 is 8 bytes

**2)** We cannot have pointers to bit field members as they may not start at a byte boundary.

|  |
| --- |
| #include <stdio.h>  struct test  {     unsigned int x: 5;     unsigned int y: 5;     unsigned int z;  };  int main()  {     test t;       // Uncommenting the following line will make     // the program compile and run     printf("Address of t.x is %p", &t.x);       // The below line works fine as z is not a     // bit field member     printf("Address of t.z is %p", &t.z);     return 0;  } |

Run on IDE

Output:

error: attempt to take address of bit-field structure member 'test::x'

**3)**It is implementation defined to assign an out-of-range value to a bit field member.

|  |
| --- |
| #include <stdio.h>  struct test  {     unsigned int x: 2;     unsigned int y: 2;     unsigned int z: 2;  };  int main()  {     test t;     t.x = 5;     printf("%d", t.x);     return 0;  } |

Run on IDE

Output:

Implementation-Dependent

**4)** In C++, we can have static members in a structure/class, but bit fields cannot be static.

|  |
| --- |
| // The below C++ program compiles and runs fine  struct test1 {     static unsigned int x;  };  int main() {  }      // But below C++ program fails in compilation as bit fields  // cannot be static  struct test1 {     static unsigned int x: 5;  };  int main() {  }  // error: static member 'x' cannot be a bit-field |

Run on IDE

**5)** Array of bit fields is not allowed. For example, the below program fails in compilation.

|  |
| --- |
| struct test  {    unsigned int x[10]: 5;  };    int main()  {    } |

Run on IDE

Output:

error: bit-field 'x' has invalid type

**Exercise:**  
Predict the output of following programs. Assume that unsigned int takes 4 bytes and long int takes 8 bytes.  
**1)**

|  |
| --- |
| #include <stdio.h>  struct test  {     unsigned int x;     unsigned int y: 33;     unsigned int z;  };  int main()  {     printf("%d", sizeof(struct test));     return 0;  } |

Run on IDE

**2)**

|  |
| --- |
| #include <stdio.h>  struct test  {     unsigned int x;     long int y: 33;     unsigned int z;  };  int main()  {     struct test t;     unsigned int \*ptr1 = &t.x;     unsigned int \*ptr2 = &t.z;     printf("%d", ptr2 - ptr1);     return 0;  } |

Run on IDE

3)

|  |
| --- |
| union test  {    unsigned int x: 3;    unsigned int y: 3;    int z;  };    int main()  {     union test t;     t.x = 5;     t.y = 4;     t.z = 1;     printf("t.x = %d, t.y = %d, t.z = %d",             t.x, t.y, t.z);     return 0;  } |

Run on IDE