Bit Magic

1. Next Power of 2

Write a function that, for a given no n, finds a number p which is greater than or equal to n and is a power of 2.

```
IP 5
OP 8

IP 17
OP 32

IP 32
OP 32
```

There are plenty of solutions for this. Let us take the example of 17 to explain some of them.

Method 1(Using Log of the number)

```
1.  Calculate Position of set bit in p(next power of 2):
    pos =  ceil(lgn)  (ceiling of log n with base 2)
2.  Now calculate p:
    p   = pow(2, pos)
```

Example

```
Let us try for 17
        pos = 5
        p   = 32
```

Method 2 (By getting the position of only set bit in result)

```
/* If n is a power of 2 then return n */
1  If (n & !(n&(n-1))) then return n
2  Else keep right shifting n until it becomes zero
    and count no of shifts
    a. Initialize: count = 0
    b. While n ! = 0
            n = n>>1
            count = count + 1
```

```
    /* Now count has the position of set bit in result */
   3  Return (1 << count)
```

Example:

```
   Let us try for 17
               count = 5
               p     = 32
```

```
unsigned int nextPowerOf2(unsigned int n)
{
  unsigned count = 0;

  /* First n in the below condition is for the case where n is 0*/
  if (n && !(n&(n-1)))
    return n;

  while( n != 0)
  {
    n  >>= 1;
    count += 1;
  }

  return 1<<count;
}

/* Driver program to test above function */
int main()
{
  unsigned int n = 0;
  printf("%d", nextPowerOf2(n));

  getchar();
  return 0;
}
```

Method 3(Shift result one by one)

Thanks to coderyogi for suggesting this method . This method is a variation of method 2 where instead of getting count, we shift the result one by one in a loop.

```
unsigned int nextPowerOf2(unsigned int n)
{
    unsigned int p = 1;
    if (n && !(n & (n - 1)))
        return n;

    while (p < n) {
        p <<= 1;
    }
    return p;
}

/* Driver program to test above function */
int main()
{
  unsigned int n = 5;
  printf("%d", nextPowerOf2(n));

  getchar();
  return 0;
}
```

Time Complexity: O(lgn)

Method 4(Customized and Fast)

```
    1. Subtract n by 1
       n = n -1


    2. Set all bits after the leftmost set bit.


    /* Below solution works only if integer is 32 bits */
                n = n | (n >> 1);
                n = n | (n >> 2);
                n = n | (n >> 4);
                n = n | (n >> 8);
                n = n | (n >> 16);
    3. Return n + 1
```

Example:

```
Steps 1 & 3 of above algorithm are to handle cases
of power of 2 numbers e.g., 1, 2, 4, 8, 16,


    Let us try for 17(10001)
    step 1
       n = n - 1 = 16 (10000)
    step 2
       n = n | n >> 1
       n = 10000 | 01000
       n = 11000
```

```
n = n | n >> 2
n = 11000 | 00110
n = 11110
n = n | n >> 4
n = 11110 | 00001
n = 11111
n = n | n >> 8
n = 11111 | 00000
n = 11111
n = n | n >> 16
n = 11110 | 00000
n = 11111

step 3: Return n+1
 We get n + 1 as 100000 (32)
```

Program:

```
# include <stdio.h>

/* Finds next power of two for n. If n itself
   is a power of two then returns n*/

unsigned int nextPowerOf2(unsigned int n)
{
    n--;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    n++;
    return n;
}

/* Driver program to test above function */
int main()
{
    unsigned int n = 5;
    printf("%d", nextPowerOf2(n));

    getchar();
    return 0;

}
```

Time Complexity: O(lgn)

References:

http://en.wikipedia.org/wiki/Power_of_2

2. Write an Efficient Method to Check if a Number is Multiple of 3

The very first solution that comes to our mind is the one that we learned in school. If sum of digits in a number is multiple of 3 then number is multiple of 3 e.g., for 612 sum of digits is 9 so it's a multiple of 3. But this solution is not efficient. You have to get all decimal digits one by one, add them and then check if sum is multiple of 3.

There is a pattern in binary representation of the number that can be used to find if number is a multiple of 3. If difference between count of odd set bits (Bits set at odd positions) and even set bits is multiple of 3 then is the number.

Example: 23 (00..10111)
1) Get count of all set bits at odd positions (For 23 it's 3).
2) Get count of all set bits at even positions (For 23 it's 1).
3) If difference of above two counts is a multiple of 3 then number is also a multiple of 3.

(For 23 it's 2 so 23 is not a multiple of 3)

Take some more examples like 21, 15, etc…

```
Algorithm: isMutlipleOf3(n)
1) Make n positive if n is negative.
2) If number is 0 then return 1
3) If number is 1 then return 0
4) Initialize: odd_count = 0, even_count = 0
5) Loop while n != 0
    a) If rightmost bit is set then increment odd count.
    b) Right-shift n by 1 bit
    c) If rightmost bit is set then increment even count.
    d) Right-shift n by 1 bit
6) return isMutlipleOf3(odd_count - even_count)
```

Proof:
Above can be proved by taking the example of 11 in decimal numbers. (In this context 11 in decimal numbers is same as 3 in binary numbers)
If difference between sum of odd digits and even digits is multiple of 11 then decimal number is multiple of 11. Let's see how.

Let's take the example of 2 digit numbers in decimal
AB = 11A -A + B = 11A + (B – A)
So if (B – A) is a multiple of 11 then is AB.

Let us take 3 digit numbers.

ABC = 99A + A + 11B – B + C = (99A + 11B) + (A + C – B)
So if (A + C – B) is a multiple of 11 then is (A+C-B)

Let us take 4 digit numbers now.
ABCD = 1001A + D + 11C – C + 999B + B – A
= (1001A – 999B + 11C) + (D + B – A -C)
So, if (B + D – A – C) is a multiple of 11 then is ABCD.

This can be continued for all decimal numbers.
Above concept can be proved for 3 in binary numbers in the same way.

Time Complexity: O(logn)

Program:

```
#include<stdio.h>

/* Fnction to check if n is a multiple of 3*/
int isMultipleOf3(int n)
{
    int odd_count = 0;
    int even_count = 0;

    /* Make no positive if +n is multiple of 3
       then is -n. We are doing this to avoid
       stack overflow in recursion*/
    if(n < 0)   n = -n;
    if(n == 0) return 1;
    if(n == 1) return 0;

    while(n)
    {
        /* If odd bit is set then
           increment odd counter */
        if(n & 1)
           odd_count++;
        n = n>>1;

        /* If even bit is set then
           increment even counter */
        if(n & 1)
            even_count++;
        n = n>>1;
    }

     return isMultipleOf3(abs(odd_count - even_count));
}

/* Program to test function isMultipleOf3 */
int main()
{
    int num = 23;
    if (isMultipleOf3(num))
        printf("num is multiple of 3");
    else
        printf("num is not a multiple of 3");
    getchar();
    return 0;
}
```

3. Write a C program to find the parity of an unsigned integer

Parity: Parity of a number refers to whether it contains an odd or even number of 1-bits. The number has “odd parity”, if it contains odd number of 1-bits and is “even parity” if it contains even number of 1-bits.
Main idea of the below solution is – Loop while n is not 0 and in loop unset one of the set bits and invert parity.

```
Algorithm: getParity(n)
1. Initialize parity = 0
2. Loop while n != 0
      a. Invert parity
            parity = !parity
      b. Unset rightmost set bit
            n = n & (n-1)
3. return parity
```

Example:

```
 Initialize: n = 13 (1101)   parity = 0

n = 13 & 12  = 12 (1100)   parity = 1
n = 12 & 11 = 8  (1000)   parity = 0
n = 8 & 7 = 0  (0000)    parity = 1
```

Program:

```
# include <stdio.h>
# define  bool int

/* Function to get parity of number n. It returns 1
   if n has odd parity, and returns 0 if n has even
   parity */
bool getParity(unsigned int n)
{
    bool parity = 0;
    while (n)
    {
        parity = !parity;
        n      = n & (n - 1);
    }
    return parity;
}

/* Driver program to test getParity() */
int main()
{
    unsigned int n = 7;
    printf("Parity of no %d = %s",  n,
             (getParity(n)? "odd": "even"));

    getchar();
    return 0;
}
```

Above solution can be optimized by using lookup table. Please refer to Bit Twiddle Hacks[1st reference] for details.

Time Complexity: The time taken by above algorithm is proportional to the number of bits set. Worst case complexity is O(Logn).

Uses: Parity is used in error detection and cryptography.

References:
http://graphics.stanford.edu/~seander/bithacks.html#ParityNaive – last checked on 30 May 2009.

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:

```c
# 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;
}
```

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

Note: Works only for positive integers.
Same concept can be used for fast multiplication by 9 or other numbers.

5. Write one line C function to find whether a no is power of two

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;
}
```

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://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 for adding this case).
Below is the implementation of this method.

```
#include<stdio.h>
#define bool int

/* Function to check if x is power of 2*/
bool isPowerOfTwo (int x)
{
  /* First x in the below expression is for the case when x is 0 */
  return x && (!(x&(x-1)));
}

/*Driver 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;
}
```

Output:

```
No
No
Yes
Yes
No
Yes
```

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

6. 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
```

Let I/P be 12 (1100)

```
Algorithm: (Example 18(010010))


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:

```
#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;
}
```

7. 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

Algorithm:
Do bitwise XOR of all the elements. Finally we get the number which has odd occurrences.

Program:

```c
#include <stdio.h>

int getOddOccurrence(int ar[], int ar_size)
{
     int i;
     int res = 0;
     for (i=0; i < ar_size; i++)
        res = res ^ ar[i];

     return res;
}

/* Diver function to test above function */
int main()
{
     int ar[] = {2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2};
     int n = sizeof(ar)/sizeof(ar[0]);
     printf("%d", getOddOccurrence(ar, n));
     return 0;
}
```

Time Complexity: O(n)

8. 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
```

```c
#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;
}
```

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;
}
```

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

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem

9. 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.

- 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;
}
```

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;
}
```

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;
}
```

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.

10. Write an Efficient C Program to Reverse Bits of a Number

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();
}
```

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;
}
```

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();
}
```

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

Method 3 – Lookup Table:
We can reverse the bits of a number in O(1) if we know the size of the number. We can implement it using look up table. Go through the below link for details. You will find some more interesting bit related stuff there.

http://www-graphics.stanford.edu/~seander/bithacks.html#BitReverseTable

11. Count set bits in an integer

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

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.

```
/* Function to get no of set bits in binary
   representation of passed binary no. */
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));
    getchar();
    return 0;
}
```

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. */
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;
}
```

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.

You can find one use of counting set bits at http://geeksforgeeks.org/?p=1465

References:

http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetNaive

12. Count number of bits to be flipped to convert A to B

Suggested by Dheeraj

Question: You are given two numbers A and B. Write a program to count number of bits needed to be flipped to convert A to B.

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.

Example:

```
A  = 1001001
B  = 0010101
a_xor_b = 1011100
No of bits need to flipped = set bit count in a_xor_b i.e. 4
```

To get the set bit count please see another post on this portal http://geeksforgeeks.org/?p=1176

13. Find the two non-repeating elements in an array of repeating elements

Asked by SG
Given an array in which all numbers except two are repeated once. (i.e. we have 2n+2 numbers and n numbers are occurring twice and remaining two have occurred once). Find those two numbers in the most efficient way.

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

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

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

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

```
Let us see an example.
   arr[] = {2, 4, 7, 9, 2, 4}
1) Get the XOR of all the elements.
     xor = 2^4^7^9^2^4 = 14 (1110)
2) Get a number which has only one set bit of the xor.
   Since we can easily get the rightmost set bit, let us use it.
     set_bit_no = xor & ~(xor-1) = (1110) & ~(1101) = 0010
   Now set_bit_no will have only set as rightmost set bit of xor.
3) Now divide the elements in two sets and do xor of
   elements in each set, and we get the non-repeating
   elements 7 and 9. Please see implementation for this
   step.
```

Implementation:

```c
#include <stdio.h>
#include <stdlib.h>

/* This finction sets the values of *x and *y to nonr-epeating
 elements in an array arr[] of size n*/
void get2NonRepeatingNos(int arr[], int n, int *x, int *y)
{
  int xor = arr[0]; /* Will hold xor of all elements */
  int set_bit_no;  /* Will have only single set bit of xor */
  int i;
  *x = 0;
  *y = 0;

  /* Get the xor of all elements */
  for(i = 1; i < n; i++)
   xor ^= arr[i];

  /* Get the rightmost set bit in set_bit_no */
  set_bit_no = xor & ~(xor-1);

  /* Now divide elements in two sets by comparing rightmost set
   bit of xor with bit at same position in each element. */
  for(i = 0; i < n; i++)
  {
    if(arr[i] & set_bit_no)
     *x = *x ^ arr[i]; /*XOR of first set */
    else
     *y = *y ^ arr[i]; /*XOR of second set*/
  }
}

/* Driver program to test above function */
int main()
{
  int arr[] = {2, 3, 7, 9, 11, 2, 3, 11};
  int *x = (int *)malloc(sizeof(int));
  int *y = (int *)malloc(sizeof(int));
  get2NonRepeatingNos(arr, 8, x, y);
  printf("The non-repeating elements are %d and %d", *x, *y);
  getchar();
}
```

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

14. 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.

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();
}
```

Please write comments if you find any bug in the above program or other ways to solve the same problem.

15. Compute the minimum or maximum of two integers without branching

On some rare machines where branching is expensive, the below obvious approach to find minimum can be slow as it uses branching.

```
/* The obvious approach to find minimum (involves branching) */
int min(int x, int y)
{
  return (x < y) ? x : y
}
```

Below are the methods to get minimum(or maximum) without using branching. Typically, the obvious approach is best, though.

Method 1(Use XOR and comparison operator)

Minimum of x and y will be

```
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.

To find the maximum, use

```
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));
}

/* Driver program to test above functions */
int main()
{
  int x = 15;
  int y = 6;
  printf("Minimum of %d and %d is ", x, y);
  printf("%d", min(x, y));
  printf("\nMaximum of %d and %d is ", x, y);
  printf("%d", max(x, y));
  getchar();
}
```

Method 2(Use subtraction and shift)
If we know that

```
INT_MIN <= (x - y) <= INT_MAX
```

, then we can use the following, which are faster because (x - y) only needs to be evaluated once.

Minimum of x and y will be

```
y + ((x - y) & ((x - y) >>(sizeof(int) * CHAR_BIT - 1)))
```

This method shifts the subtraction of x and y by 31 (if size of integer is 32). If (x-y) is smaller than 0, then (x -y)>>31 will be 1. If (x-y) is greater than or equal to 0, then (x - y)>>31 will be 0.
So if x >= y, we get minimum as y + (x-y)&0 which is y.
If x < y, we get minimum as y + (x-y)&1 which is x.

Similarly, to find the maximum use

```
x - ((x - y) & ((x - y) >> (sizeof(int) * CHAR_BIT - 1)))
```

```
#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 maximum of x and y*/
int max(int x, int y)
{
  return x - ((x - y) & ((x - y) >>
            (sizeof(int) * CHAR_BIT - 1)));
}

/* Driver program to test above functions */
int main()
{
  int x = 15;
  int y = 6;
  printf("Minimum of %d and %d is ", x, y);
  printf("%d", min(x, y));
  printf("\nMaximum of %d and %d is ", x, y);
  printf("%d", max(x, y));
  getchar();
}
```

Note that the 1989 ANSI C specification doesn't specify the result of signed right-shift, so above method is not portable. If exceptions are thrown on overflows, then the values of x and y should be unsigned or cast to unsigned for the subtractions to avoid unnecessarily throwing an exception, however the right-shift needs a signed operand to produce all one bits when negative, so cast to signed there.

Source:
http://graphics.stanford.edu/~seander/bithacks.html#IntegerMinOrMax

16. Compute modulus division by a power-of-2-number

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

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

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

Now doing it is so easy, guess it….

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

```
#include<stdio.h>

/* This function will return n % d.
   d must be one of: 1, 2, 4, 8, 16, 32, … */
unsigned int getModulo(unsigned int n, unsigned int d)
{
  return ( n & (d-1) );
}

/* Driver program to test above function */
int main()
{
  unsigned int n = 6;
  unsigned int d = 4; /*d must be a power of 2*/
  printf("%u moduo %u is %u", n, d, getModulo(n, d));

  getchar();
  return 0;
}
```

References:
http://graphics.stanford.edu/~seander/bithacks.html#ModulusDivisionEasy

Please write comments if you find any bug in the above program/algorithm or other ways to solve the same problem.

17. 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 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;
}
```

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);
}
```

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 for more details about the above two methods.

Please write comments if you find any of the above explanations/algorithms incorrect, or a better ways to solve the same problem.

References:
http://graphics.stanford.edu/~seander/bithacks.html#IntegerAbs

18. Find whether a given number is a power of 4 or not

Asked by Ajay

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.

```c
#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();
}
```

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 for suggesting the approach and providing the code.

```c
#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();
}
```

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

19. 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;
}
```

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem

20. 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;
}
```

2. Another way of doing this could be (8*x – x)/2 (See below code). Thanks to ajaym for suggesting this.

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

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem

21. 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 0011000000). 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;
}
```

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;
}
```

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 comment for implementation of decrement.
Note that this method works only if the numbers are stored in 2's complement form.

Thanks to Venki for suggesting this method.

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem

22. Optimization Techniques | Set 1 (Modulus)

Modulus operator is costly.

The modulus operator (%) in various languages is costly operation. Ultimately every operator/operation must result in processor instructions. Some processors won't have modulus instruction at hardware level, in such case the compilers will insert stubs (predefined functions) to perform modulus. It impacts performance.

There is simple technique to extract remainder when a number is divided by another number (divisor) that is power of 2? A number that is an exact power of 2 will have only one bit set in it's binary representation. Consider the following powers of 2 and their binary representations

2 – 10

4 – 100

8 – 1000

16 – 10000

Note those zeros in red color, they contribute to remainder in division operation. We can get mask for those zeros by decrementing the divisor by 1.

Generalizing the above pattern, a number that can be written in 2^n form will have only one bit set followed by *n* zeros on the right side of 1. When a number (N) divided by (2^n), the bit positions corresponding to the above mentioned *zeros* will contribute to the remainder of division operation. An example can make it clear,

```
N = 87 (1010111 - binary form)

N%2 = N & (2-1) = 1010111 & 1 = 1 = 1

N%4 = N & (4-1) = 1010111 & 11 = 11 = 3
```

```
N%8 = N & (8-1) = 1010111 & 111 = 111 = 7

N%16 = N & (16-1) = 1010111 & 1111 = 111 = 7

N%32 = N & (32-1) = 1010111 & 11111 = 10111 = 23
```

Modulus operation over exact powers of 2 is simple and faster bitwise ANDing. This is the reason, programmers usually make buffer length as powers of 2.

Note that the technique will work only for divisors that are powers of 2.

An Example:

Implementation of circular queue (ring buffer) using an array. Omitting one position in the circular buffer implementation can make it easy to distinguish between *full* and *empty* conditions. When the buffer reaches SIZE-1, it needs to wrap back to initial position. The wrap back operation can be simple AND operation if the buffer size is power of 2. If we use any other size, we would need to use modulus operation.

Note:

Per experts comments, premature optimization is an evil. The optimization techniques provided are to fine tune your code after finalizing design strategy, algorithm, data structures and implementation. We recommend to avoid them at the start of code development. Code readability is key for maintenance.

Thanks to Venki for writing the above article. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

23. Next higher number with same number of set bits

Given a number x, find next number with same number of 1 bits in it's binary representation.

For example, consider x = 12, whose binary representation is 1100 (excluding leading zeros on 32 bit machine). It contains two logic 1 bits. The next higher number with two logic 1 bits is 17 (10001_2).

Algorithm:

When we observe the binary sequence from 0 to $2^n - 1$ (n is # of bits), right most bits (least significant) vary rapidly than left most bits. The idea is to find right most string of 1's in x, and shift the pattern to right extreme, except the left most bit in the pattern. Shift

the left most bit in the pattern (omitted bit) to left part of x by one position. An example makes it more clear,

```
x = 156
```

10

```
x = 10011100
```

(2)

```
10011100
00011100 - right most string of 1's in x
00000011 - right shifted pattern except left most bit ------> [A]
00010000 - isolated left most bit of right most 1's pattern
00100000 - shiftleft-ed the isolated bit by one position ------> [B]
10000000 - left part of x, excluding right most 1's pattern ------> [C]
10100000 - add B and C (OR operation) ------> [D]
10100011 - add A and D which is required number 163
```

(10)

After practicing with few examples, it easy to understand. Use the below given program for generating more sets.

Program Design:

We need to note few facts of binary numbers. The expression x & -x will isolate right most set bit in x (ensuring x will use 2's complement form for negative numbers). If we add the result to x, right most string of 1's in x will be reset, and the immediate '0' left to this pattern of 1's will be set, which is part [B] of above explanation. For example if x = 156, x & -x will result in 00000100, adding this result to x yields 10100000 (see part D). We left with the right shifting part of pattern of 1's (part A of above explanation).

There are different ways to achieve part A. Right shifting is essentially a division operation. What should be our divisor? Clearly, it should be multiple of 2 (avoids 0.5 error in right shifting), and it should shift the right most 1's pattern to right extreme. The expression (x & -x) will serve the purpose of divisor. An EX-OR operation between the number X and expression which is used to reset right most bits, will isolate the rightmost 1's pattern.

A Correction Factor:

Note that we are adding right most set bit to the bit pattern. The addition operation causes a shift in the bit positions. The weight of binary system is 2, one shift causes an increase by a factor of 2. Since the increased number (*rightOnesPattern* in the code) being used twice, the error propagates twice. The error needs to be corrected. A right shift by 2 positions will correct the result.

The popular name for this program is **s**ame **n**umber **o**f **o**ne **b**its.

```
#include<iostream>

using namespace std;

typedef unsigned int uint_t;

// this function returns next higher number with same number of set bi
uint_t snoob(uint_t x)
{

  uint_t rightOne;
  uint_t nextHigherOneBit;
  uint_t rightOnesPattern;

  uint_t next = 0;

  if(x)
  {

    // right most set bit
    rightOne = x & -(signed)x;

    // reset the pattern and set next higher bit
    // left part of x will be here
    nextHigherOneBit = x + rightOne;

    // nextHigherOneBit is now part [D] of the above explanation.

    // isolate the pattern
    rightOnesPattern = x ^ nextHigherOneBit;

    // right adjust pattern
    rightOnesPattern = (rightOnesPattern)/rightOne;

    // correction factor
    rightOnesPattern >>= 2;

    // rightOnesPattern is now part [A] of the above explanation.

    // integrate new pattern (Add [D] and [A])
    next = nextHigherOneBit | rightOnesPattern;
  }

  return next;
}

int main()
{
  int x = 156;
  cout<<"Next higher number with same number of set bits is "<<snoob(x

  getchar();
  return 0;
}
```

Usage: Finding/Generating subsets.

Variations:

1. Write a program to find a number immediately smaller than given, with same number of logic 1 bits? (Pretty simple)
2. How to count or generate the subsets available in the given set?

References:

1. A nice presentation here.
2. Hackers Delight by Warren (An excellent and short book on various bit magic algorithms, a must for enthusiasts)
3. C A Reference Manual by Harbison and Steele (A good book on standard C, you can access code part of this post here).

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

24. 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 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 2^{32} 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;
}
```

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

25. A Boolean Array Puzzle

Input: A array arr[] of two elements having value 0 and 1

Output: Make both elements 0.

Specifications: Following are the specifications to follow.
1) It is guaranteed that one element is 0 but we do not know its position.
2) We can't say about another element it can be 0 or 1.
3) We can only complement array elements, no other operation like and, or, multi, division, …. etc.
4) We can't use if, else and loop constructs.
5) Obviously, we can't directly assign 0 to array elements.

There are several ways we can do it as we are sure that always one Zero is there.
Thanks to devendraiiit for suggesting following 3 methods.

Method 1

```
void changeToZero(int a[2])
{
   a[ a[1] ] = a[ !a[1] ];
}

int main()
{
   int a[] = {1, 0};
   changeToZero(a);

   printf(" arr[0] = %d \n", a[0]);
   printf(" arr[1] = %d ", a[1]);
   getchar();
   return 0;
}
```

Method 2

```
void changeToZero(int a[2])
{
    a[ !a[0] ] = a[ !a[1] ]
}
```

Method 3
This method doesn't even need complement.

```
void changeToZero(int a[2])
{
    a[ a[1] ] = a[ a[0] ]
}
```

Method 4
Thanks to purvi for suggesting this method.

```
void changeToZero(int a[2])
{
  a[0] = a[a[0]];
  a[1] = a[0];
}
```

There may be many more methods.

Source: http://geeksforgeeks.org/forum/topic/google-challenge

Please write comments if you find the above codes incorrect, or find other ways to solve the same problem.

26. 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;
}
```

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 (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;
}
```

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 and Vignesh 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;
}
```

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

27. 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 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 no
        x = x ^ y;

        // Carry is shifted by one so that adding it to x gives the re
        y = carry << 1;
    }
    return x;
}

int main()
{
    printf("%d", Add(15, 32));
    return 0;
}
```

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);
}
```

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

28. 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.

Examples:

```
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. 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, unsigne
{
    /* 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;
}
```

Output:

```
Result = 7
```

Following is a shorter implementation of the same logic

```
int swapBits(unsigned int x, unsigned int p1, unsigned int p2, unsigne
{
    /* xor contains xor of two sets */
    unsigned int xor = ((x >> p1) ^ (x >> p2)) & ((1U << n) - 1);

    /* To swap two sets, we need to again XOR the xor with original se
    return x ^ ((xor << p1) | (xor << p2));
}
```

References:

Swapping individual bits with XOR

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

29. 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
```

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;
}
```

Output:

```
Total set bit count is 6
```

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
#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,.
    // 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;
}
```

```
Total set bit count is 35
```

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

Thanks to agatsu and IC for suggesting this solution.

See this for another solution suggested by Piyush Kapoor.

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

30. 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;
}
```

Output:

```
Signs are opposite
```

Source: Detect if two integers have opposite signs

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);
}
```

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);
}
```

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.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

31. 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
```

We can use sorting to do it in O(nLogn) time. We can also use hashing, but the worst case time complexity of hashing may be more than O(n) and hashing 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.

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'.

```
#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 b
           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 '

           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 '

           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;
}
```

Output:

```
2
```

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

Following is another O(n) time complexity and O(1) extra space method suggested by *aj*. We can sum the bits in same positions for all the numbers and take modulo with 3. The bits for which sum is not multiple of 3, are the bits of number with single occurrence.
Let us consider the example array {5, 5, 5, 8}. The 101, 101, 101, 1000
Sum of first bits%3 = (1 + 1 + 1 + 0)%3 = 0;
Sum of second bits%3 = (0 + 0 + 0 + 0)%0 = 0;
Sum of third bits%3 = (1 + 1 + 1 + 0)%3 = 0;
Sum of fourth bits%3 = (1)%3 = 1;
Hence number which appears once is 1000

```
#include <stdio.h>
#define INT_SIZE 32
```

```
int getSingle(int arr[], int n)
{
    // Initialize result
    int result = 0;

    int x, sum;

    // Iterate through every bit
    for (int i = 0; i < INT_SIZE; i++)
    {
      // Find sum of set bits at ith position in all
      // array elements
      sum = 0;
      x = (1 << i);
      for (int j=0; j< n; j++ )
      {
          if (arr[j] & x)
            sum++;
      }

      // The bits with sum not multiple of 3, are the
      // bits of element with single occurrence.
      if (sum % 3)
        result |= x;
    }

    return result;
}
```

```
// Driver program to test above function
int main()
{
    int arr[] = {12, 1, 12, 3, 12, 1, 1, 2, 3, 2, 2, 3, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("The element with single occurrence is %d ",
            getSingle(arr, n));
    return 0;
}
```

```
7
```

This article is compiled by **Sumit Jain** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

32. Binary representation of a given number

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

Source: Microsoft Interview Set-3

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);
}
```

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);
}
```

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

33. Write your own strcmp that ignores cases

Write a modified strcmp function which ignores cases and returns -1 if s1 < s2, 0 if s1 = s2, else returns 1. For example, your strcmp should consider "GeeksforGeeks" and "geeksforgeeks" as same string.

Source: Microsoft Interview Set 5

Following solution assumes that characters are represented using ASCII representation, i.e., codes for ‘a’, ‘b’, ‘c’, … ‘z’ are 97, 98, 99, … 122 respectively. And codes for ‘A’, “B”, ‘C’, … ‘Z’ are 65, 66, … 95 respectively.

Following are the detailed steps.
1) Iterate through every character of both strings and do following for each character.
…**a)** If str1[i] is same as str2[i], then continue.
…**b)** If inverting the 6th least significant bit of str1[i] makes it same as str2[i], then continue. For example, if str1[i] is 65, then inverting the 6th bit will make it 97. And if str1[i] is 97, then inverting the 6th bit will make it 65.
…**c)** If any of the above two conditions is not true, then break.
2) Compare the last (or first mismatching in case of not same) characters.

```
#include <stdio.h>

/* implementation of strcmp that ingnores cases */
int ic_strcmp(char *s1, char *s2)
{
    int i;
    for (i = 0; s1[i] && s2[i]; ++i)
    {
        /* If characters are same or inverting the 6th bit makes them
        if (s1[i] == s2[i] || (s1[i] ^ 32) == s2[i])
           continue;
        else
           break;
    }

    /* Compare the last (or first mismatching in case of not same) cha
    if (s1[i] == s2[i])
        return 0;
    if ((s1[i]|32) < (s2[i]|32)) //Set the 6th bit in both, then compa
        return -1;
    return 1;
}

// Driver program to test above function
int main(void)
{
    printf("ret: %d\n", ic_strcmp("Geeks", "apple"));
    printf("ret: %d\n", ic_strcmp("", "ABCD"));
    printf("ret: %d\n", ic_strcmp("ABCD", "z"));
    printf("ret: %d\n", ic_strcmp("ABCD", "abcdEghe"));
    printf("ret: %d\n", ic_strcmp("GeeksForGeeks", "gEEksFORGeEKs"));
    printf("ret: %d\n", ic_strcmp("GeeksForGeeks", "geeksForGeeks"));
    return 0;
}
```

Output:

```
ret: 1
ret: -1
ret: -1
ret: -1
ret: 0
ret: 0
```

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

34. 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”.

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 t
// same length by aadding leading 0s in the smaller string. Returns th
// 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 addit
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
```

```
    // 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) | (firstB
    }

    // 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;
}
```

Output:

```
Sum is 1100101
```

This article is compiled by **Ravi Chandra Enaganti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

35. 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.

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 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;
}
```

Output:

```
43
```

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

36. 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

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
// 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 htt
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 number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 12;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 128;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    return 0;
}
```

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 number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 12;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 128;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    return 0;
}
```

Output:

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

We can also use log base 2 to find the position. Thanks to Arunkumar 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 number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 12;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 128;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    return 0;
}
```

Output:

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

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

37. 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.

```
   X = 101001 = 41
   Y = 101010 = 42
-------------
      1010010
    101001
+ 101001
-------------
  11010111010 = 1722
```

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

$$X = Xl*2^{n/2} + Xr \quad [Xl \text{ and } Xr \text{ contain leftmost and rightmost } n/2 \text{ bits of } X]$$

$$Y = Yl*2^{n/2} + Yr \quad [Yl \text{ and } Yr \text{ contain leftmost and rightmost } n/2 \text{ bits of } Y]$$

The product XY can be written as following.

$$XY = (Xl*2^{n/2} + Xr)(Yl*2^{n/2} + Yr)$$
$$= 2^n XlYl + 2^{n/2}(XlYr + XrYl) + XrYr$$

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

$$XY = 2^n XlYl + 2^{n/2} * [(Xl + Xr)(Yl + Yr) - XlYl - XrYr] + XrYr$$

With above trick, the recurrence becomes T(n) = 3T(n/2) + O(n) and solution of this recurrence is $O(n^{1.59})$.

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.

$$XY = 2^{2ceil(n/2)} XlYl + 2^{ceil(n/2)} * [(Xl + Xr)(Yl + Yr) - XlYl - XrYr] + XrYr$$

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

Following is C++ implementation of above algorithm.

```
// C++ implementation of Karatsuba algorithm for bit string multiplica
#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 t
// 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 addit
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&c
    }

    // 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 retur
// 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"));
}
```

Output:

```
120
60
30
```

```
10
0
49
9
```

Time Complexity: Time complexity of the above solution is $O(n^{1.59})$.

Time complexity of multiplication can be further improved using another Divide and Conquer algorithm, fast Fourier transform. We will soon be discussing fast Fourier transform as a separate post.

Exercise
The above program returns a long int value and will not work for big strings. Extend the above program to return a string instead of a long int value.

References:
Wikipedia page for Karatsuba algorithm
Algorithms 1st Edition by Sanjoy Dasgupta, Christos Papadimitriou and Umesh Vazirani
http://courses.csail.mit.edu/6.006/spring11/exams/notes3-karatsuba
http://www.cc.gatech.edu/~ninamf/Algos11/lectures/lect0131.pdf

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

38. 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;
}
```

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;
}
```

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;
}
```

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;
}
```

Output:

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

Swapping a variable with itself may needed in many standard algorithms. For example see this implementation of QuickSort 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;
}
```

Output:

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

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

39. 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 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;
}
```

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.

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

40. 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.

```
#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;
}
```

Output:

```
7
```

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

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

41. Swap two nibbles in a byte

A 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.

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;
}
```

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.

This article is contributed by **Anuj Garg**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

42. 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. We start from leftmost and rightmost bits and compare bits one by one. If we find a mismatch, then return false.

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 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;
}
```

Output:

```
1
1
```

This article is contributed by **Saurabh Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

43. 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;
}
```

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

  n^2 = (2*x)^2 = 4*x^2
If n is odd, it can be written as
  n = 2*x + 1

  n^2 = (2*x + 1)^2 = 4*x^2 + 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;
}
```

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).

This article is contributed by **Ujjwal Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

44. Subtract two numbers without using arithmetic operators

Write a function subtract(x, y) that returns x-y where x and y are integers. The function should not use any of the arithmetic operators (+, ++, –, -, .. etc).

The idea is to use bitwise operators. Addition of two numbers has been discussed using Bitwise operators. Like addition, the idea is to use subtractor logic.

The truth table for the half subtractor is given below.

```
X   Y      Diff     Borrow
0   0   0   0
0   1   1   1
1   0   1   0
1   1   0   0
```

From the above table one can draw the Karnaugh map for “difference” and “borrow”.

So, Logic equations are:

Diff = $y \oplus x$
Borrow = $\overline{x} \cdot$ y

Source: Wikipedia page for subtractor

Following is C implementation based on above equations.

```
#include<stdio.h>

int subtract(int x, int y)
{
    // Iterate till there is no carry
    while (y != 0)
    {
        // borrow contains common set bits of y and unset
        // bits of x
        int borrow = (~x) & y;

        // Subtraction of bits of x and y where at least
        // one of the bits is not set
        x = x ^ y;

        // Borrow is shifted by one so that subtracting it from
        // x gives the required sum
        y = borrow << 1;
    }
    return x;
}

// Driver program
int main()
{
    int x = 29, y = 13;
    printf("x - y is %d", subtract(x, y));
    return 0;
}
```

Output:

```
x - y is 16
```

Following is recursive implementation for the same approach.

```
#include<stdio.h>

int subtract(int x, int y)
{
    if (y == 0)
        return x;
    return subtract(x ^ y, (~x & y) << 1);
}

// Driver program
int main()
{
    int x = 29, y = 13;
    printf("x - y is %d", subtract(x, y));
    return 0;
}
```

Output:

```
x - y is 16
```

This article is contributed **Dheeraj**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above