1. **Find the element that appears once from the list where all other elements appear thrice  
     
   Solution:** Run a loop for all elements in the array, at the end of every iteration, maintain following two values:

**Ones:** The bits which have appeared 1st time, 4th time, 7th time etc

**Twos:** The bits which have appeared for 2nd time, 5th time, 8th time etc.

And we do a thing for every element. Now, if current element is being considered, ones and twos keep track of the bits which have appeared 1st time, 4th time, 7th time etc or 2nd time, 5th time, 8th time etc in elements with arr[0] to arr[current-1]

Now, if a bit is set in current element and the same bit is set in ones, we unset the bit from **ones**, and set it in **twos**

**n=current\_number&ones**

**ones=ones&(~n)**

**twos=twos|n**

And, if a bit is set in current element and the same bit is not set is ones, there are two possibilities.

**First,** the bit is appearing for the first time in element range element[0..curr\_index]

**Second, the bit is appearing for the** 3rd time, 6th time, 9th time in the range element[0..curr\_index]

In both cases, the **first decision step** is common. We will **set the bit in ones. (necessary step. Otherwise, the next step of detecting the bit which is going to be set for 3rd time, 6th time, 9th time, etc…will fail)**

Now, if it is appeared 3rd time, 6th time, 9th time etc, we need to take an additional step. Since, it is of no interest in the problem’s context (it is unnecessary data) we will remove it. Now, if a bit is appearing for 3rd time, 6th time, 9th time etc, according to the way it will be **present in both ones and twos. Now, we have to find it and we need to remove it from both ones and twos.**

**Bit\_collection\_appeared\_for\_3rd\_time=ones & twos;**

**Ones=ones & (~bit\_collection\_appeared\_for\_third\_time);**

**Twos=twos & (~bit\_collection\_appeared\_for\_third\_time);**

At the end, ones will contain the answer.

1. **Detect If two integers have Opposite signs.**If xor of them is -ve.
2. **Find a number which appeared once when all other numbers appeared twice/even times in the list.**We can do it by xor.  
     
   **4. swap two/three numbers without using temporary variables.**

a=a^b;

b=a^b;

a=a^b;

a=a^b^c;

b=a^b^c;

c=a^b^c;

a=a^b^c;

But, there is a problem is in this approach, too. If, same variable’s reference is sent to both a and b.

**5.Check For Integer Overflow:**

If INT\_MAX-a>b

a+b will definitely cause integer overflow.

1. **Multiply a number with 3.5:**(x<<3-x)>>1
2. **Multiply a number with 7:**(x<<3-x)
3. **Add 1 to a given number:**

**First way:**~n=-(n+1)   
-(~n)=n+1  
  
~n is bitwise negation. Or, simply flipping 1’s to 0’s and 0’s to 1’s.  
  
**Second Way:  
  
This approach is found by observation.** find the rightmost unset bit. Flip all the bits which lie in the right side of the rightmost unset bit. Finally, flip the rightmost unset bit, too.

1. **Decrease 1 from a given number**like, a way was found for increasing a number by 1 from observing the pattern. There’s a simple way for decreasing 1 from a given number.
2. **Find if a number is power of 4 or not.  
     
   Only 1 bit needs to be set.**

And, the number of 0’s in the right side of the only set bit should be of even count.

1. **Check whether a number is power of 2 or not.**If(n&(n-1)) is 0.

**12.Count the number of set bits in a number?**int count=0;  
while(n!=0)

{  
 n=n&(n-1);

count++;

}

**13.Add two bit strings:**

Digital logic is required.

1. **Calculate xor of 0’s and 1’s in the binary representation:**Let’s say, the number is an integer. (32 bit)  
   Now, the number of set bits can be calculated from this:  
   int count=0;  
   while(n!=0)  
   {  
    n=n&(n-1);  
    count++;

}  
  
Now, count will contain the number of set bits.

The number of unset bits will be 32-count.  
Last step is xoring them.

1. **Check if two numbers are equal without arithmetic and comparison operators:**by xoring them
2. **Calculate xor of 1 to n.**there is a relation with mod 4. find it.
3. **Convert a binary number string to octal.**First step is, if total length is not divisible by 3, add zeros at the left side.  
   Then , we need to read it in size of 3 and convert it to octal representation.  
     
   Note: in c, octal number starts with 0. and hexadecimal with 0x. though it is irrelevant in current context.
4. **Check if in a binary array the number represented by a sub array is odd or even:**last bit.
5. **Convert decimal fraction to a binary number.**

I can do that.

1. **Sum of numbers with exactly two bits set.**Two loops. (32 \*32)
2. **Josephus Problem with bit magic**If the total number of persons is 2^n+l then solution is 2l+1  
     
   Now, the primary task is finding l. Because, once, we found l, the answer is l<<1|1.

**Now, l is** (2n+l)&(~(1<<position of leftmost bit in 0 index)  
  
Or, (2n+l)^(1<<position of leftmost bit in 0 index)

Now, logarithm can help to find the position of leftmost bit of a number.

1. **Position of rightmost set bit**

int left\_most\_pos;  
while(true)  
{

Left\_most\_pos=floor(log2(n));

If(n&(1<<left\_most\_pos)!=0)

{

n=n^(1<<left\_most\_pos);

}

else

{

break;

}

}

Or, log2(n&(~n+1))  
  
this will also return the rightmost bit’s set position.

1. **How to reset or unset the rightmost set bit**n=n&(n-1)
2. **Find position of only set bit:**log base 2 probably offers the best solution.  
   Otherwise, we can count based on right shifting.
3. **Swap two nibbles in a byte:**I can do that.

**26.How to turn off a particular bit in a number**if(n&(1<<pos\_of\_given\_bit\_in\_0\_indexing))  
{  
 n=n^(1<<pos\_of\_given\_bit\_in\_0\_indexing);

}

**27.Write your own strcmp:**if(str1[i]==str2[i])||(str1[i]^32==str2[i]))  
  
**28. Multiply a number with 10 without using multiplication operator:**

I can do that.

1. **Check if a number has bits in alternate position.**A simple approach is to find it’s binary equivalent and check it’s bits.  
   Or, we can do both I.e. finding the binary representation and checking it’s bits simultaneously.

**30.Equal sum and xor.**We know,   
  
a+b=a^b+2\*(a&b)  
  
Now, a^b+2\*(a&b)=a^b  
  
that implies that a&b must be 0.  
  
that reduces the problem that for a given a we need to find the number of b’s for which a&b=0.

Now, we need to calculate the number of unset bits in b or number of set bits in a, Now, all set bits in a must be reset. Now, for left positions, we can have either 1 or 0 in the b. Hence, the total number of b is 2number\_of\_unset\_bits\_in\_a.

1. **Russian Peasant method for multiplication:**

#include <iostream>

using namespace std;

// A method to multiply two numbers using Russian Peasant method

unsigned int russianPeasant(unsigned int a, unsigned int b)

{

int res = 0; // initialize result

// While second number doesn't become 1

while (b > 0)

{

// If second number becomes odd, add the first number to result

if (b & 1)

res = res + a;

// Double the first number and halve the second number

a = a << 1;

b = b >> 1;

}

return res;

//even if it is 2’s power, it will become odd at the end, that’s when the a will be added

//but, be careful about this method, this can cause problem

}

1. **Toggle case of a string using bitwise operator.**xor it with 32.
2. **Toggle bits of a given range.**Xor based operation. I can do that.
3. **Unset bit of a given range:**
4. **Find the largest number with n set bits and m unset bits**
5. **Find the smallest number with n unset bits and m set bits:**I can do these three problems.
6. **One’s complement and two’s complement:  
     
   One’s Complement:**Invert all bits. Each 1 becomes 0, and each 0 becomes 1. **Two’s Complement:**the important aspect of two’s complement is that it automatically includes the concept of signed bit.  
     
   How to convert it in two’s complement:  
     
   **Begin with original binary value:  
     
   Find the one’s complement  
     
   Add 1 to the one’s complement**
7. **How to convert a binary string in it’s two’s complement:**

string findTwoscomplement(string str)

{

int n = str.length();

// Traverse the string to get first '1' from

// the last of string

int i;

for (i = n-1 ; i >= 0 ; i--)

if (str[i] == '1')

break;

// If there exists no '1' concatenate 1 at the

// starting of string

if (i == -1)

return '1' + str;

// Continue traversal after the position of

// first '1'

for (int k = i-1 ; k >= 0; k--)

{

//Just flip the values

if (str[k] == '1')

str[k] = '0';

else

str[k] = '1';

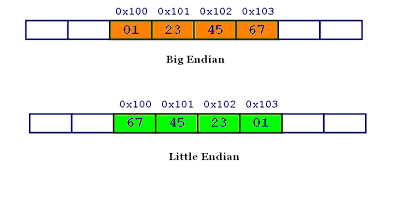
}

// return the modified string

return str;

}

1. **Big endianness and little endianness:**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.

**  
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;

}

**The quickest way to check endianness:**

#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;

}