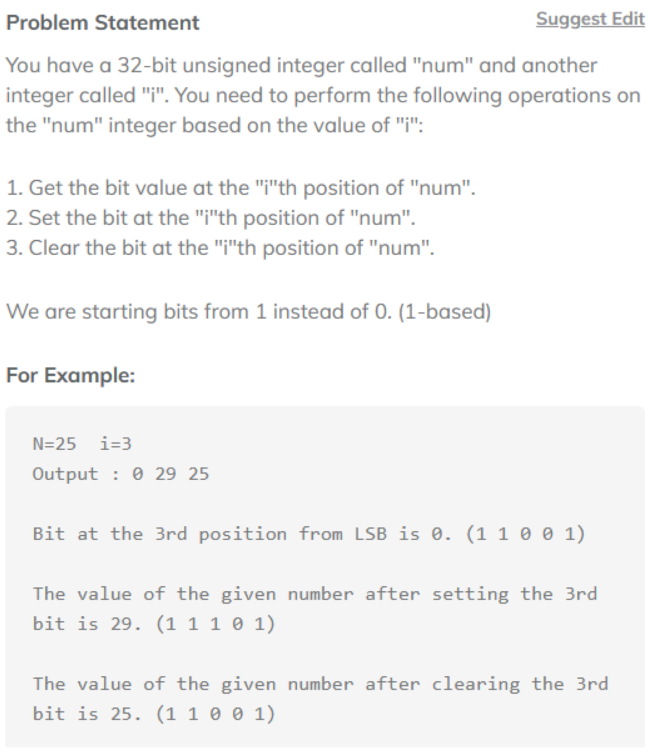
BIT MANIPULATION:

Imp points:

1. **To unset the right most set bit to unset, n & n-1**
2. **To set the right most unset bit to set , n | n+1 .**
3. **To find whether a element is power of 2 or not, n & n-1 ==0 , then yes.**
4. **2^n can be written as 1<<n (1 left shift n)**
5.

Q:

Perform following functions:



Problem Statement Suggest Edit

You have a 32-bit unsigned integer called "num" and another integer called "i". You need to perform the following operations on the "num" integer based on the value of "i":

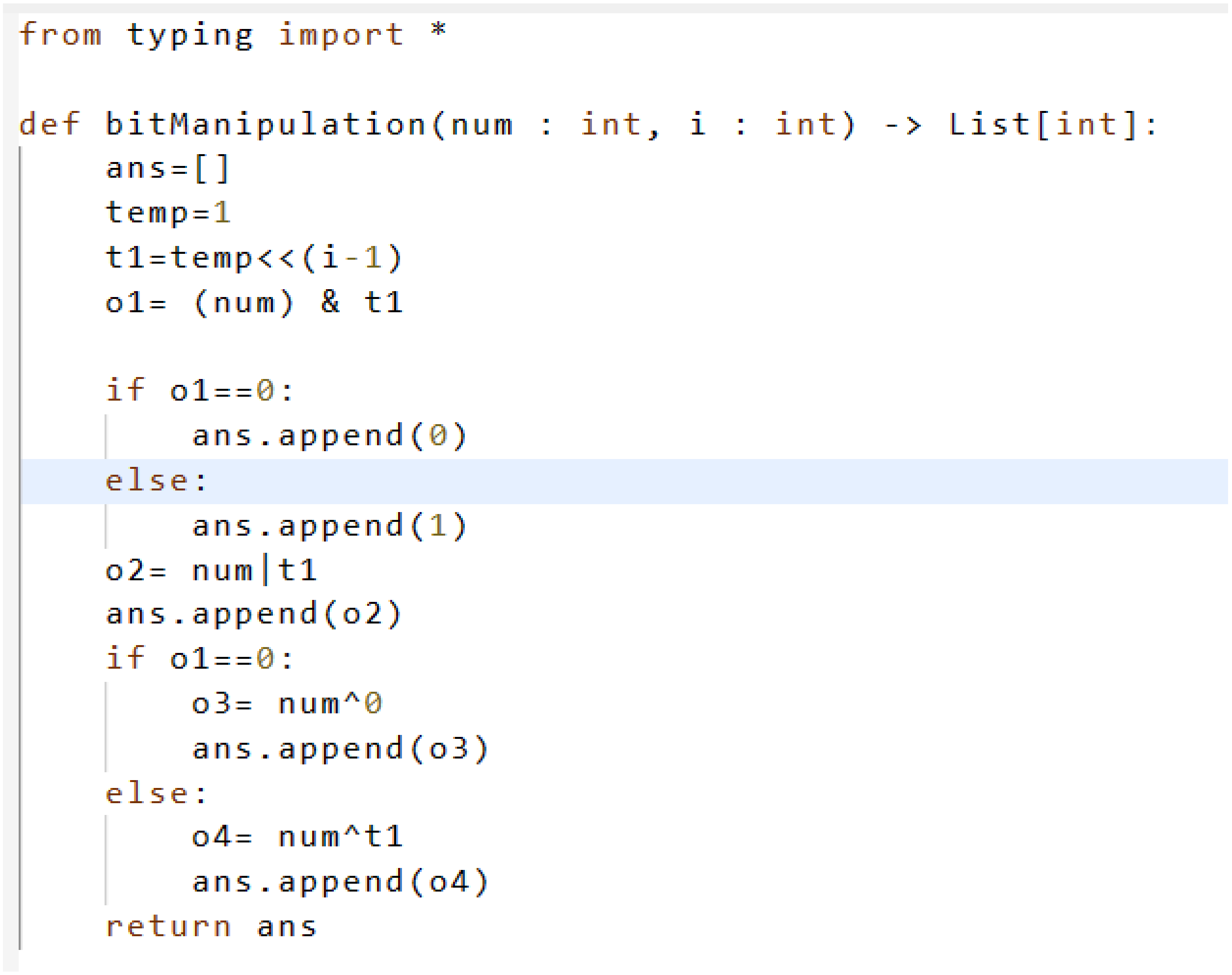
1. Get the bit value at the "i"th position of "num".
2. Set the bit at the "i"th position of "num".
3. Clear the bit at the "i"th position of "num".

We are starting bits from 1 instead of 0. (1-based)

For Example:

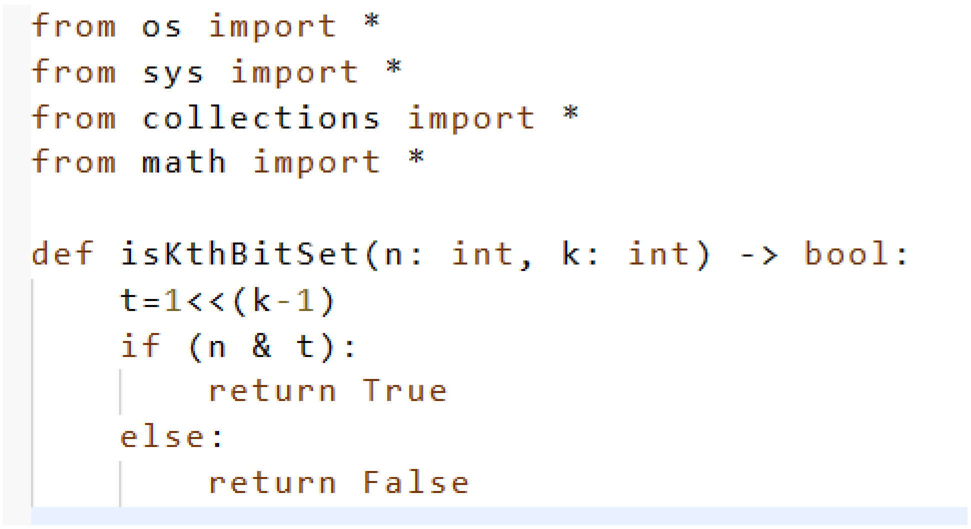
```
N=25  i=3
Output : 0 29 25

Bit at the 3rd position from LSB is 0. (1 1 0 0 1)

The value of the given number after setting the 3rd
bit is 29. (1 1 1 0 1)

The value of the given number after clearing the 3rd
bit is 25. (1 1 0 0 1)
```

TIME COMPLEXITY: O(1)



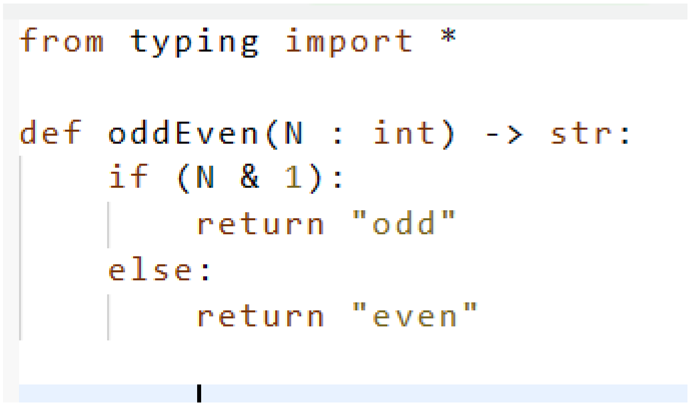
```python
from typing import *

def bitManipulation(num : int, i : int) -> List[int]:
    ans=[]
    temp=1
    t1=temp<<(i-1)
    o1= (num) & t1

    if o1==0:
        ans.append(0)
    else:
        ans.append(1)
    o2= num|t1
    ans.append(o2)
    if o1==0:
        o3= num^0
        ans.append(o3)
    else:
        o4= num^t1
        ans.append(o4)
    return ans
```

Q2: Check whether K-th bit is set or not:



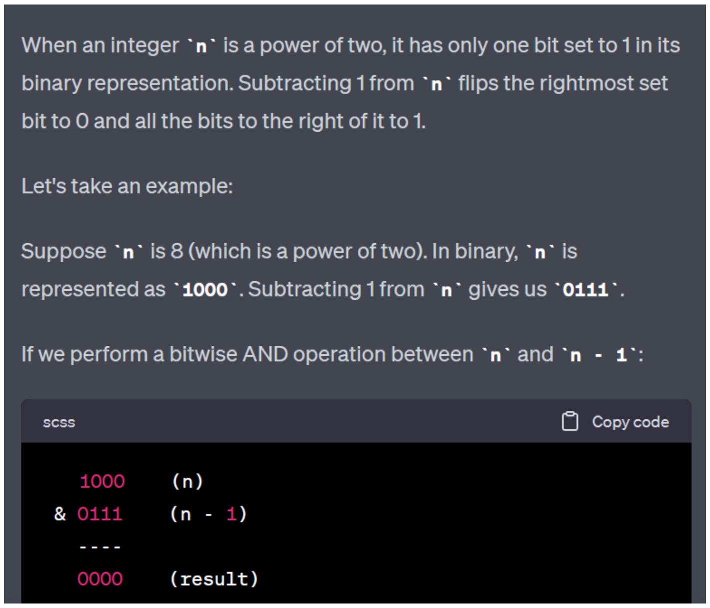
```python
from os import *
from sys import *
from collections import *
from math import *

def isKthBitSet(n: int, k: int) -> bool:
    t=1<<(k-1)
    if (n & t):
        return True
    else:
        return False
```

Q3: Check even or odd:



```python
from typing import *

def oddEven(N : int) -> str:
    if (N & 1):
        return "odd"
    else:
        return "even"
```

Q: Power of 2:



When an integer `n` is a power of two, it has only one bit set to 1 in its binary representation. Subtracting 1 from `n` flips the rightmost set bit to 0 and all the bits to the right of it to 1.

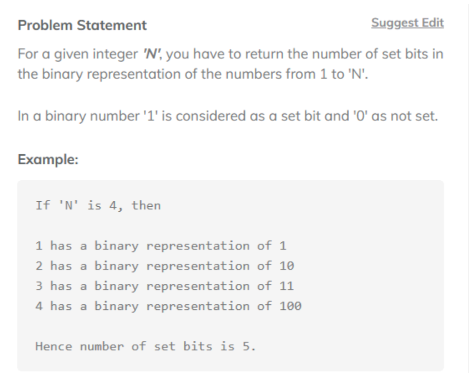
Let's take an example:

Suppose `n` is 8 (which is a power of two). In binary, `n` is represented as `1000`. Subtracting 1 from `n` gives us `0111`.

If we perform a bitwise AND operation between `n` and `n - 1`:

```
  1000    (n)
& 0111    (n - 1)
  ----
  0000    (result)
```

Q:



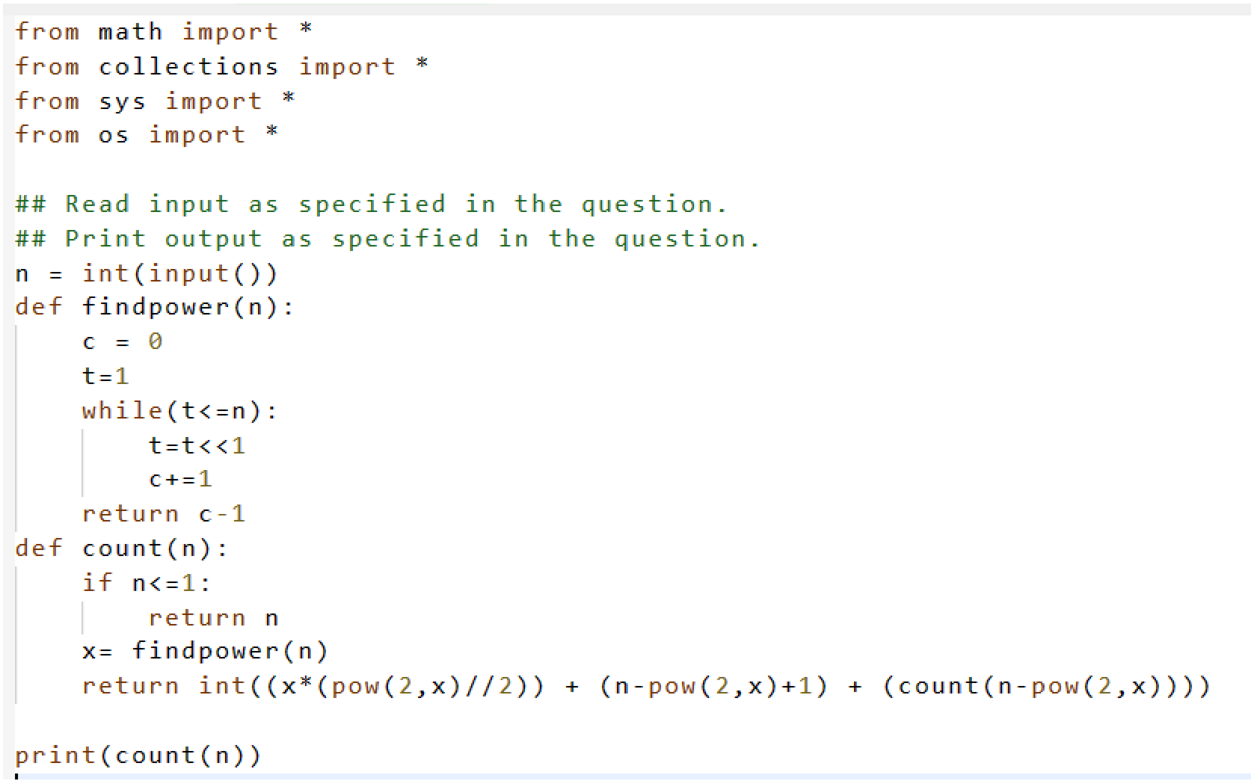
Problem Statement

For a given integer *'N'*, you have to return the number of set bits in the binary representation of the numbers from 1 to 'N'.

In a binary number '1' is considered as a set bit and '0' as not set.

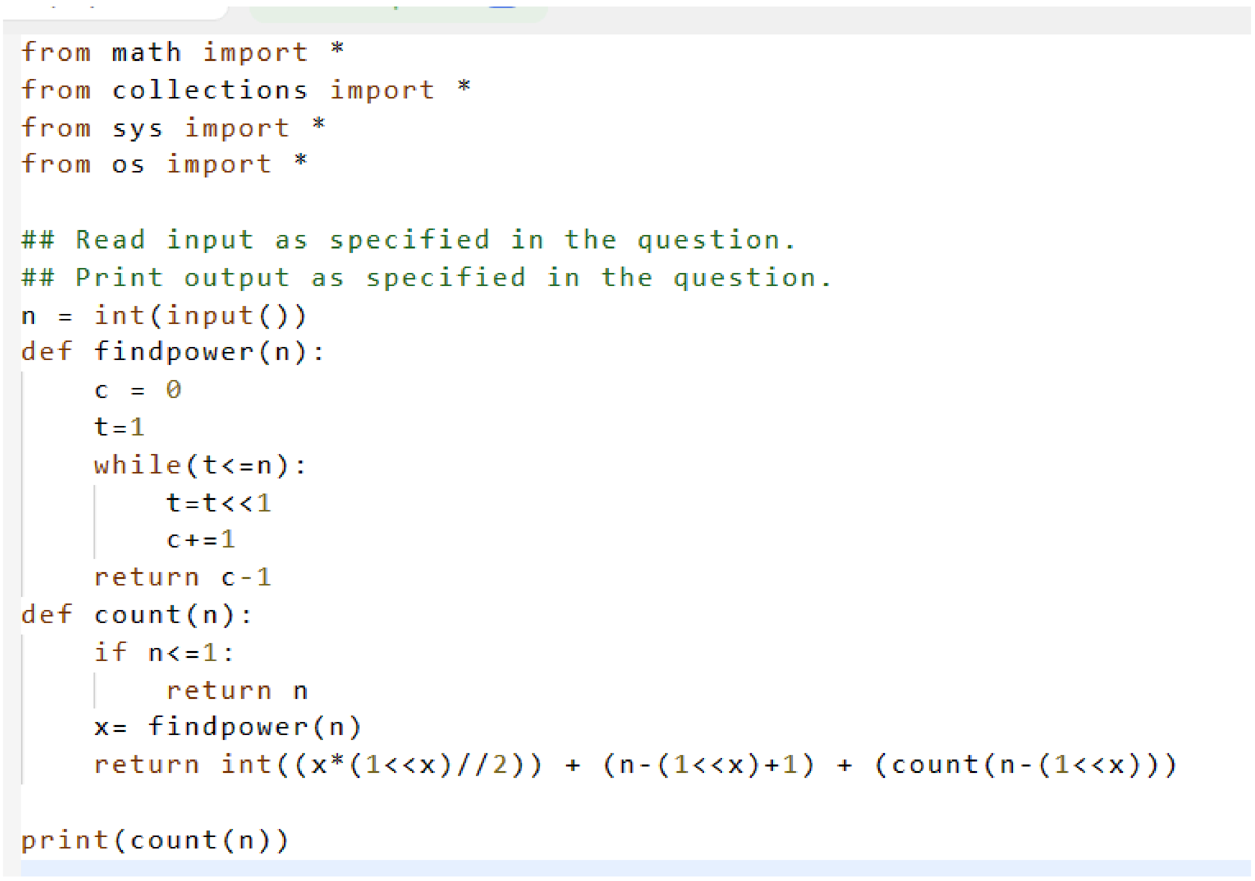
Example:

```
If 'N' is 4, then

1 has a binary representation of 1
2 has a binary representation of 10
3 has a binary representation of 11
4 has a binary representation of 100

Hence number of set bits is 5.
```

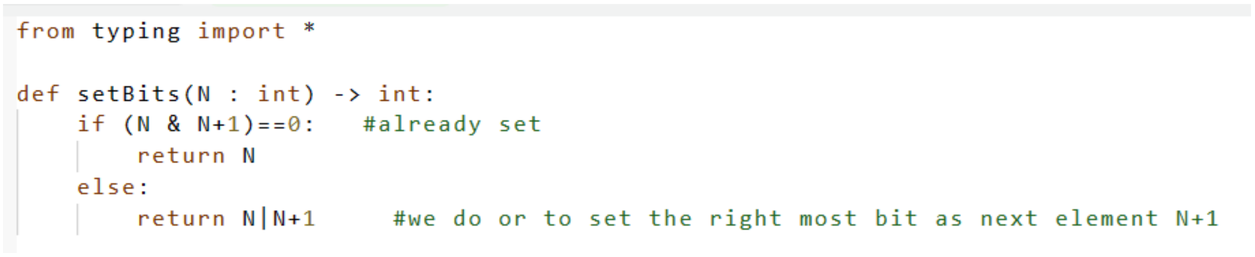


```
from math import *
from collections import *
from sys import *
from os import *

## Read input as specified in the question.
## Print output as specified in the question.
n = int(input())
def findpower(n):
    c = 0
    t=1
    while(t<=n):
        t=t<<1
        c+=1
    return c-1
def count(n):
    if n<=1:
        return n
    x= findpower(n)
    return int((x*(pow(2,x)//2)) + (n-pow(2,x)+1) + (count(n-pow(2,x))))

print(count(n))
```

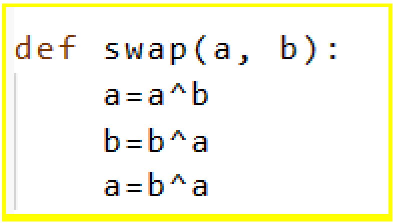
TIP : IF we need to find (2^n). ie, 2 ^ 3 , then left shift 1 three times, then it is equal to 8 . ‘
For ex: 1<<3= 1000 =8


```python
from math import *
from collections import *
from sys import *
from os import *

## Read input as specified in the question.
## Print output as specified in the question.
n = int(input())
def findpower(n):
    c = 0
    t=1
    while(t<=n):
        t=t<<1
        c+=1
    return c-1
def count(n):
    if n<=1:
        return n
    x= findpower(n)
    return int((x*(1<<x)//2)) + (n-(1<<x)+1) + (count(n-(1<<x)))

print(count(n))
```

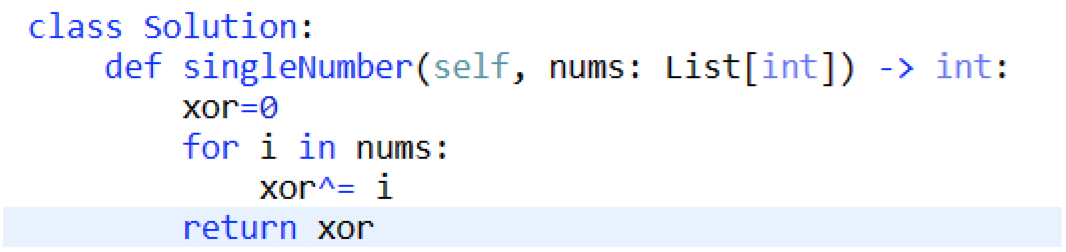
Q: Rightmost Unset bit:


```python
from typing import *

def setBits(N : int) -> int:
    if (N & N+1)==0:   #already set
        return N
    else:
        return N|N+1     #we do or to set the right most bit as next element N+1
```

Q: Swap two number:


```python
def swap(a, b):
    a=a^b
    b=b^a
    a=b^a
```

Q: To check which no occurs once , given only one number occurs one and rest occurs two times.

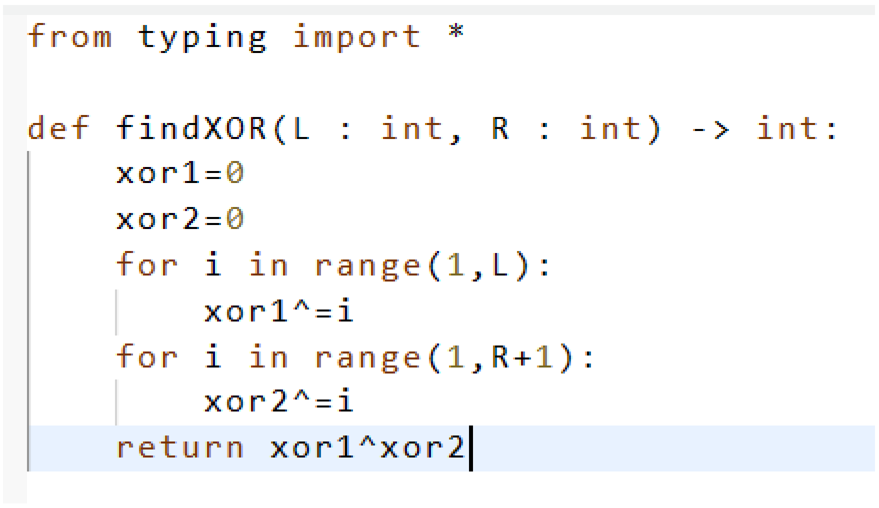


```python
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        xor=0
        for i in nums:
            xor^= i
        return xor
```

Q:L to R XOR:

XOR of digit from 3 to 7 ie, FIND 3^4^5^6^7=

(3,7): 3= 1^2
7= 1^2^3^4^5^6^7
When we do 3^7= 1^2^1^2^3^4^5^6^7= ans



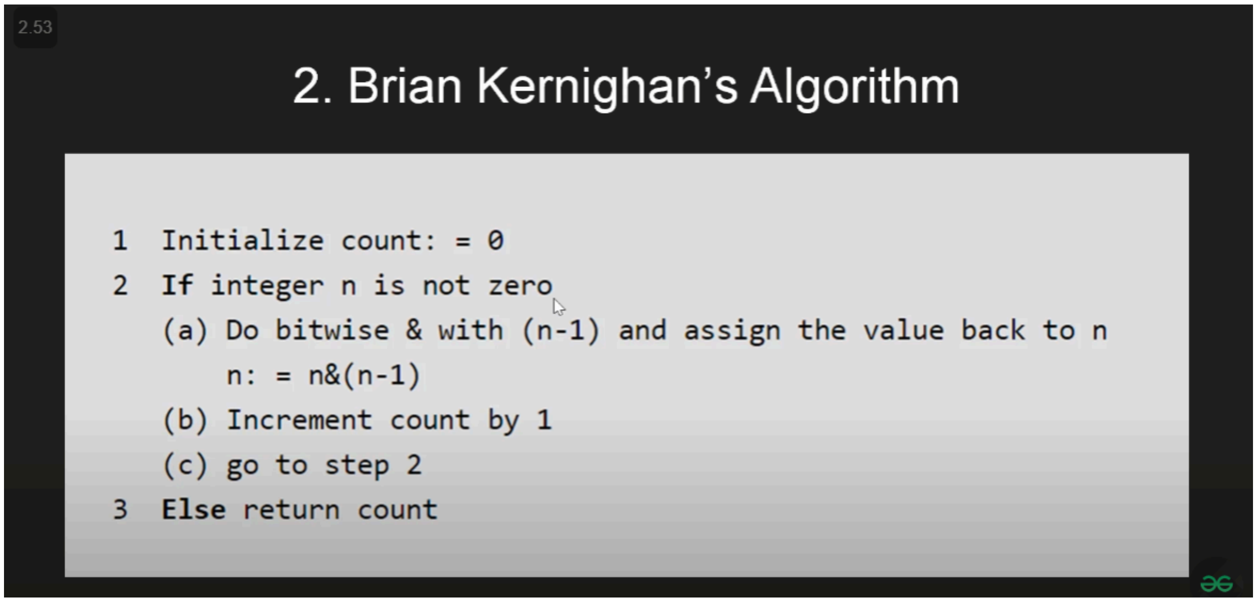
```python
from typing import *

def findXOR(L : int, R : int) -> int:
    xor1=0
    xor2=0
    for i in range(1,L):
        xor1^=i
    for i in range(1,R+1):
        xor2^=i
    return xor1^xor2
```

Q: Find the element which are present only once, there are exactly two element which are present only once, and rest present twice.

TC: O(N)
SC=O(1)

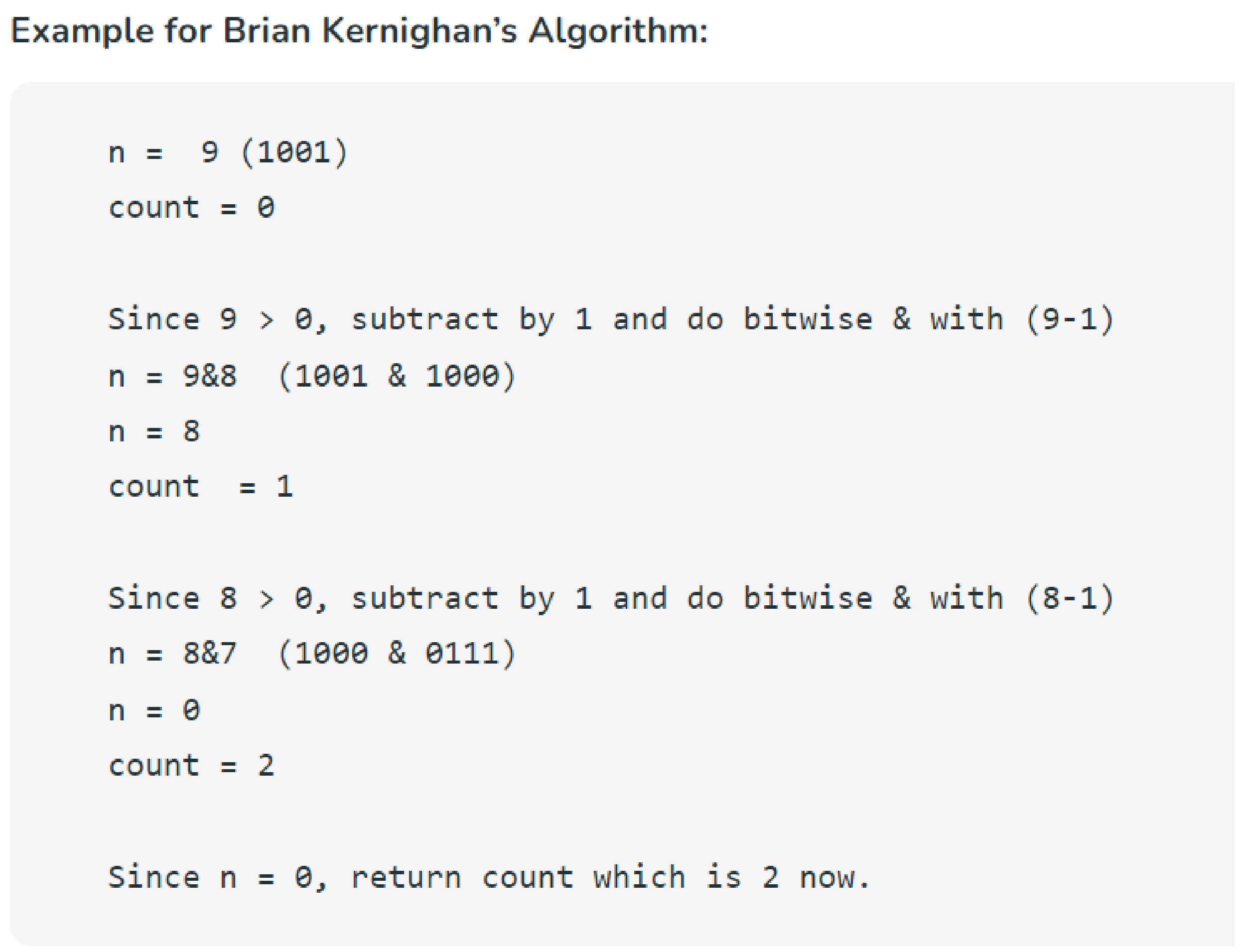
```
class Solution:
    def singleNumber(self, nums: List[int]) -> List[int]:
        xor = 0
        for i in nums:
            xor^= i
        #nums = [1,2,1,3,2,5]
        #when we do xor of every element then the two number which remain left. lets say here 3^5 remains
        #find the right most differ bit of 3^5 , as we do xor then the right most bit which are differ having
value set. so we will find out which right most bit is set, startng from index 0 from right

        cnt=0
        while(xor):
            if xor&1:
                break      #index 0 bit is set so count is returned as 0 . ie, 0th bit set
            else:
                cnt+=1
                xor=xor>>1
        n=len(nums)

        #now partition it in two different part one which is having same the right most bit as same.
        xor1=0
        xor2=0
        for i in range(n):
            if(nums[i] & (1<<cnt)):        #to check weather the bit at cnt position is set or not
                xor1^=nums[i]
            else:
                xor2^=nums[i]
        return [xor1,xor2]
```

Count set bits in an integer:



2. Brian Kernighan's Algorithm

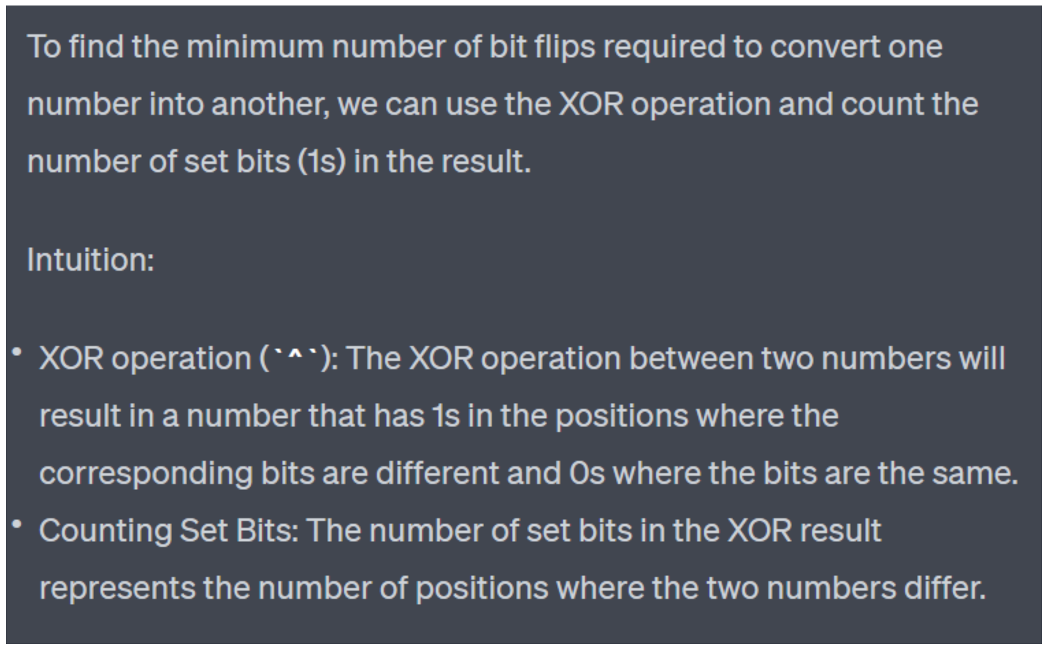
```
1  Initialize count: = 0
2  If integer n is not zero
   (a) Do bitwise & with (n-1) and assign the value back to n
       n: = n&(n-1)
   (b) Increment count by 1
   (c) go to step 2
3  Else return count
```



Example for Brian Kernighan's Algorithm:

```
n =  9 (1001)
count = 0

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

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

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

Q Minimum Bit Flips to Convert Number(based on above algo)



To find the minimum number of bit flips required to convert one number into another, we can use the XOR operation and count the number of set bits (1s) in the result.

Intuition:

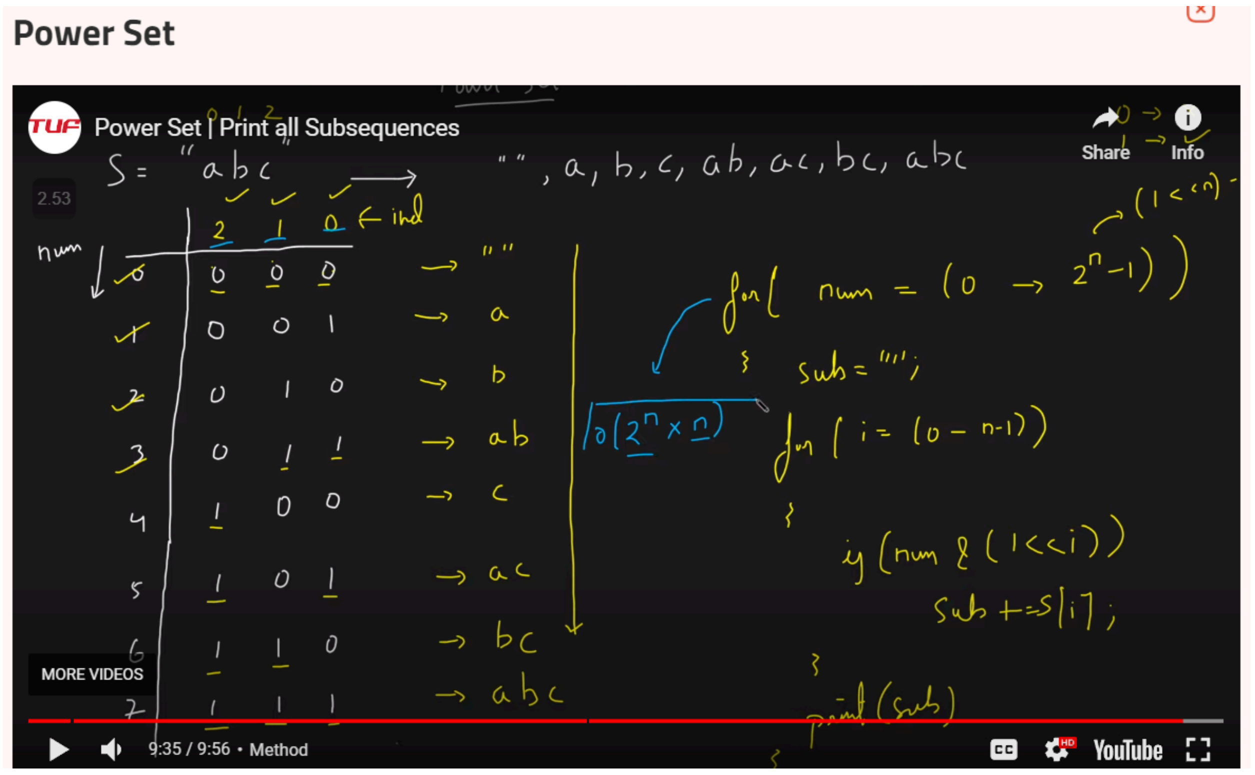
- XOR operation (`^`): The XOR operation between two numbers will result in a number that has 1s in the positions where the corresponding bits are different and 0s where the bits are the same.
- Counting Set Bits: The number of set bits in the XOR result represents the number of positions where the two numbers differ.

```python
class Solution:
    def minBitFlips(self, start: int, goal: int) -> int:
        xor= start^ goal   #find how many bits are different.
        #count no of sets bit and return
        cnt=0
        while(xor):
            if xor & 1:       #to check the that bit at last is set or not
                cnt+=1
            xor=xor>>1        #we right shift the number.
        return cnt
```

2nd solution:

```python
class Solution:
    def minBitFlips(self, start: int, goal: int) -> int:
        n= start^goal
        count=0

        while(n):
            n = n&(n-1)
            count+=1
        return count
```

Q Power set:

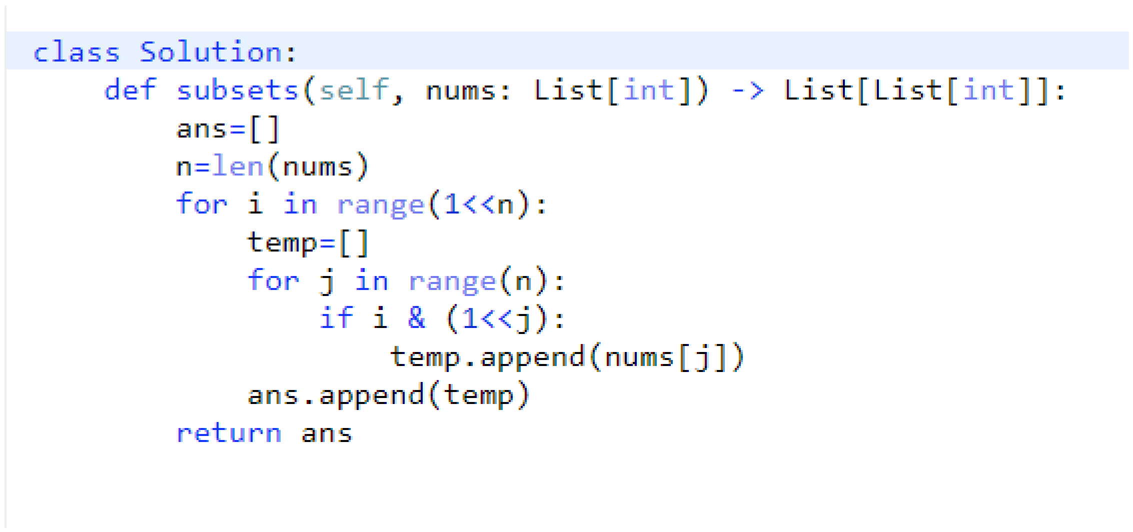


S = "abc" → "", a, b, c, ab, ac, bc, abc

num	2	1	0 ← ind	
0	0	0	0	→ ""
1	0	0	1	→ a
2	0	1	0	→ b
3	0	1	1	→ ab
4	1	0	0	→ c
5	1	0	1	→ ac
6	1	1	0	→ bc
7	1	1	1	→ abc

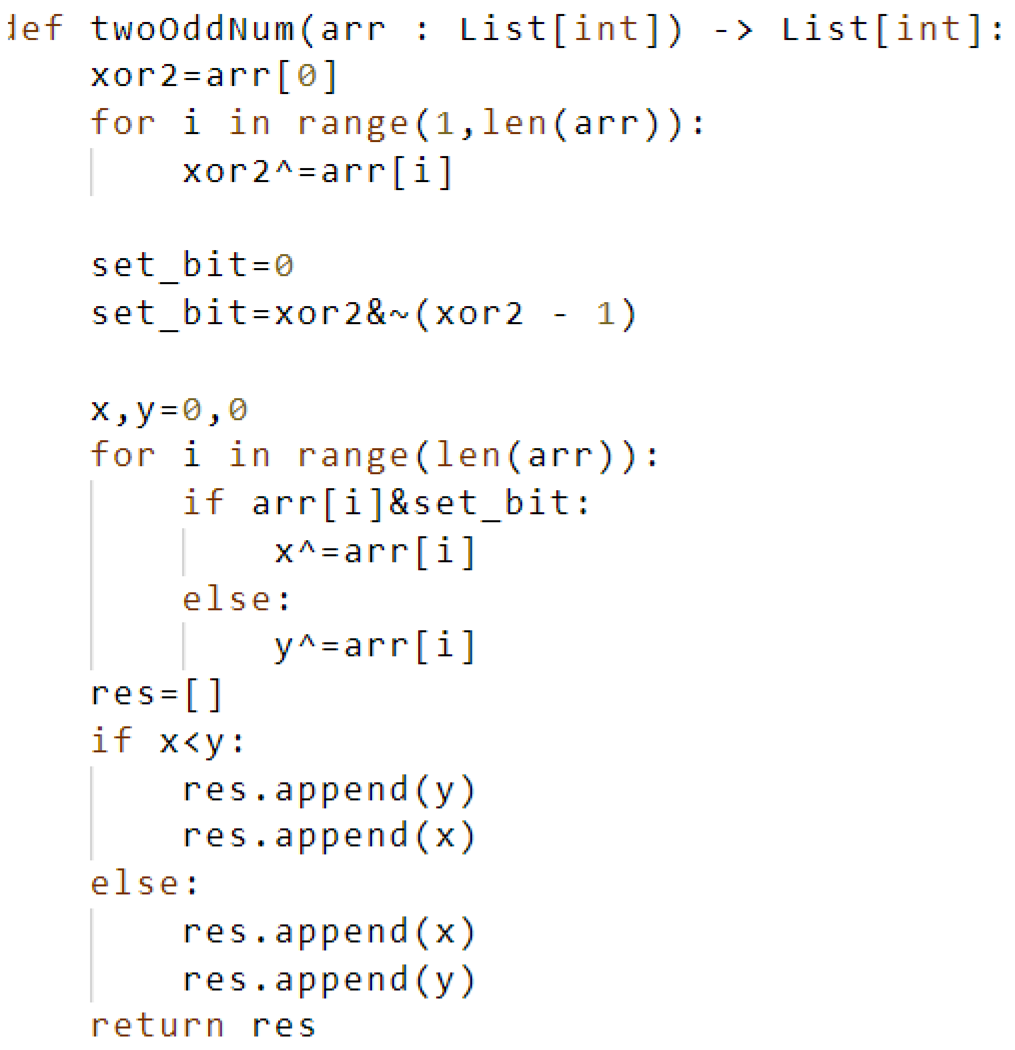
$O(2^n \times n)$

```
for( num = (0 → 2^n -1) )      (1<<n)
{   sub = "";
    for ( i = (0 - n-1))
    {
        if (num & (1<<i))
            sub += S[i];
    }
    print(sub)
}
```



```python
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        ans=[]
        n=len(nums)
        for i in range(1<<n):
            temp=[]
            for j in range(n):
                if i & (1<<j):
                    temp.append(nums[j])
            ans.append(temp)
        return ans
```

Q **Two Numbers With Odd Occurrences:**



```
def twoOddNum(arr : List[int]) -> List[int]:
    xor2=arr[0]
    for i in range(1,len(arr)):
        xor2^=arr[i]

    set_bit=0
    set_bit=xor2&~(xor2 - 1)

    x,y=0,0
    for i in range(len(arr)):
        if arr[i]&set_bit:
            x^=arr[i]
        else:
            y^=arr[i]
    res=[]
    if x<y:
        res.append(y)
        res.append(x)
    else:
        res.append(x)
        res.append(y)
    return res
```