1 - Bit Manipulation

Convert to base

1780. Check if Number is a Sum of Powers of Three

https://leetcode.com/problems/check-if-number-is-a-sum-of-powers-of-three/description/

The problem is asking if a number if a sum of **distinct powers of three**. If you've only read the title, the problem would have been much harder lolz. Making it a sum of **distinct powers of three** makes everything simpler.

Think of a brute force solution. Given a number, say 12, how can we determine if it's a sum of three? The simplest way is just to try it out with pen & paper. Let's write down the list of power of three:

$3^0, 3^1, 3^2, 3^3, 3^4, 3^5 \dots$
or:
$1, 3, 9, 27, 81, 243, 729$...

Assuming 1 is part of the sum, then the rest (12-1=11) should be a sum of other powers of 3. Since it has to be sum of distinct numbers, we cannot use 1 again. Let's try 3, then (11-3=8) has to be a sum of powers of 3. 8 < 9 so that doesn't work.

Let's try again, starting from 3. (12-3=9) is already a power of three. Hence a solution is
$12 = 3 + 9 = 3^1 + 3^2$
We can try the same for 91 and find $91 = 3^0 + 3^2 + 3^4$.

Soon, with only a few examples, you'll see this is not a good solution. The constraint states that $1 \leq n \leq 10^7$. This means the list of powers of three could go up to X where
$3^X = 10^7 \rightarrow X = log_3(10^7) = 4771212$.

Now that's > 4M numbers in the list. Each time we scrap off one, we still have a lot to check.

A simpler solution is to think in terms of binary representation, but **this time using base 3**.

Recall that if a number m in base 10 is represented in base b as:
$m_{10} = (a_n a_{n-1} a_{n-2} \dots a_0)_b$
then:
$m = b^0 * a_0 + b^1 * a_1 + b^2 * a_2 + \dots + b^n * a_n$
In our case, b = 3 so:
$m = 3^0 * a_0 + 3^1 * a_1 + 3^2 * a_2 + \dots + 3^n * a_n$

We can see that in order for m to be a sum of distinct powers of 3, all the digits a should be either 0 or 1.

Solution: Convert the number m from base 10 to base 3. Check all the digits $a_0, a_1, a_2, \ldots a_n$. Note that for base 3 these digits can only take the values of {0, 1, 2}. If any of them is 2, that means n contains the sum of 2 same powers of 3 → all digits must be either 0 or 1.

Now we know the solution is through converting n from base 10 to base 3. Let's recall some knowledge on base conversion. We shall get started from base 2: binary.

Binary conversion

We have:

$m_{10} = (a_n a_{n-1} a_{n-2} \ldots a_0)_2$

i.e.:

$m = 2^0 * a_0 + 2^1 * a_1 + 2^2 * a_2 + \ldots + 2^n * a_n$

where $a_i \in \{0, 1\}$

We can perform binary conversion by finding all a_i one step at a time, starting from a_0. We can rewrite m as:

$m = a_0 + 2 * (2^0 * a_1 + 2^1 * a_2 + \ldots + 2^{n-1} * a_n)$

Because a_0 could be either 0 or 1, and the 2^{nd} component of the right-hand-side (RHS) is a multiple of 2, a_0 must be the remainder of the division of n by 2. So we know $a_0 = m\%2$. Now we can rewrite the above equation as:

$(m - a_0)/2 = 2^0 * a_1 + 2^1 * a_2 + \ldots + 2^{n-1} * a_n$

While the numbers have changed, you can see above equation as the same form but this time n is replaced by $(m - a_0)/2$. (Note that since a_0 is the remainder of m divided by 2, $(m - a_0)$ must be divisible by 2.). So we can repeat above step to get a_1, rewrite the equation and continue to find a_3, and so on.

So when do we stop? Remember that if n is 0, it could be represented as an infinite number of 0, i.e.:

$0_{10} = (0)_2$

$0_{10} = (00)_2$

$0_{10} = (000)_2$

$0_{10} = (000\ldots\ldots\ldots 0)_2$

In fact, for any number, we can put an infinite number of 0 in front of it and it won't change the value. So if n reaches 0, we should stop otherwise we'll just keep generating 0 without changing anything.

Let's put above process into code:

```
def convert2binary(n):
    # constraint n >= 0, i'm not doing 2-complement here
    if n == 0:
        return [0]
    ret = [] # ret is short for return
    while n > 0:
        a0 = n % 2
        ret.append(a0)
        n = (n - a0) // 2 # we do integer division to keep n an int, note
that (n-a0) is always divisible by 2
    return ret
```

Let's double check:

```
In [17]: def convert2binary(n):
    ...:     if n == 0:
    ...:         return [0]
    ...:     ret = []
    ...:     while n > 0:
    ...:         a0 = n % 2
    ...:         ret.append(a0)
    ...:         n = (n - a0) // 2
    ...:     return ret
    ...:

In [18]: convert2binary(0)
Out[18]: [0]

In [19]: convert2binary(1)
Out[19]: [1]

In [20]: convert2binary(2)
Out[20]: [0, 1]

In [21]: convert2binary(3)
Out[21]: [1, 1]
```

```
In [22]: convert2binary(4)
Out[22]: [0, 0, 1]

In [23]: convert2binary(5)
Out[23]: [1, 0, 1]

In [24]: convert2binary(6)
Out[24]: [0, 1, 1]

In [25]: convert2binary(7)
Out[25]: [1, 1, 1]

In [26]: convert2binary(100)
Out[26]: [0, 0, 1, 0, 0, 1, 1]
```

Note that we append to `ret` the least significant bits (or right-most bits) so the order is reversed, e.g. for 100 we should read the result right to left as $100_{10} = (1100100)_2$.

We can improve the algorithm further by noticing that we can simply replace `n = (n - a0) // 2` with `n //= 2` . Recall `//` is integer division, so 10//3 = 3 not 3.333. Think about it, `a0` is either 0 or 1, doesn't matter if it is subtracted from n, `(n - a0) //2` is equivalent to `n//2` . For example if n = 4 → a0 = 0 → (4-0)/2 = 2 same as 4/2. If n = 5 → a0 = 1 → (5-1)/2=2 same as 5//2=2. So:

```
def convert2binary(n):
    if n == 0:
        return [0]
    ret = []
    while n > 0:
        ret.append(n % 2)
        n //= 2
    return ret
```

Base-3 conversion

The above code should work for any base conversion from base 10 (if base > 10 we'll need to convert a0 to some string, e.g. for hex it's ABCDEF remember?). A simple code change for base 3 is:

```
def convert2base3(n):
    if n == 0:
        return [0]
    ret = []
```

```
    while n > 0:
        ret.append(n % 3)
        n //= 3
    return ret
```

Some tests:

```
In [27]: def convert2base3(n):
    ...:     if n == 0:
    ...:         return [0]
    ...:     ret = []
    ...:     while n > 0:
    ...:         ret.append(n % 3)
    ...:         n //= 3
    ...:     return ret
    ...:

In [28]: convert2base3(9)
Out[28]: [0, 0, 1]

In [29]: convert2base3(20)
Out[29]: [2, 0, 2]
```

We can verify this result a bit by manually converting to base 3. The table format below is a good method to use with pen/paper.

m	next_m = m // 3	LSB = (m % 3)
9	3	0
3	1	0
1	0	1

So $9_{10} = (100)_3 = 1 * 3^2 + 0 * 3^1 + 0 * 3^0$

Or convert 20 to base 3:

m	next_m = m // 3	LSB = (m % 3)
20	6	2
6	2	0
2	0	2

Verify: $20_{10} = (202)_3 = 2 * 3^2 + 0 * 3^1 + 2 * 3^0$

Basically, to convert a number m **>= 0** from base 10 to base b, we keep calculating the remainder (m % b) → m = m//b → repeat until m is 0.

```
def convert10toB(n):
    # constraint: n >= 0. I don't know how to do 2-complement for base b.
    if n == 0:
        return [0]
    ret = [] # ret stands for return, note you can't use return for
variable name, it's a keyword
    while n:
        ret.append(n % b)
        n //= b # same as n = n // b, but //= does it in-place. You'll see
this term some time in classes
    # ret[0] is for b^0, ret[1] for b^1... and so on
    return ret
```

Note that in above code, we change `when n > 0:` to `when n:` . Recall that the when loop takes in a condition, either True or False. If True it goes inside the loop and False it gets out. In Python, any number if consider True unless it is 0.

```
def testNumber(n):
    if n:
        print(f"{n} considered True")
    else:
        print(f"{n} considered False")
```

```
In [31]: testNumber(-100)
-100 considered True

In [32]: testNumber(-1)
-1 considered True

In [33]: testNumber(0)
0 considered False

In [34]: testNumber(0.000000)
0.0 considered False
```

```
In [35]: testNumber(1)
1 considered True

In [36]: testNumber(1.234)
1.234 considered True

In [37]: testNumber(1991)
1991 considered True
```

Is it a sum of distinct powers of three?

Adapting from base conversion code to our problem, we have:

```
class Solution:
    def checkPowersOfThree(self, n):
        if n == 0:
            return False
        while n:
            x = n % 3
            if x not in [0, 1]:
                return False
            n = n // 3
        return True
```

We first check the edge case of 0 which returns False since sum of powers of 3 must be positive. Then we convert n to base 3. During conversion, we check if the digits are either 0 or 1. If not, n is definitely not sum of **distinct** powers of 3 so we return False. If False is never returned, it means n gotta be a sum of distinct powers of three, so at the end of the function we return True.

Note this is a good technique to remember: sometimes we look for the opposite: is n a sum of distinct powers of 3? → check if it is not. You will see this approach appears again and again in many problems.

Time complexity:

The time complexity is calculated based on the input n, and we often care about the worst-case scenario when it takes the most of time. A simple rough tip on calculating time complexity based on the code is every basic computation is considered 1 time unit: multiplication, division, shift, if-else...

Inside the while loop, `x = n % 3` is 1 time unit (I'll call it TU from now on). The `if x not in...` can be considered 1 TU, but beware if checking if an item is in a list takes `len(the-list)` TU as it literally goes through the least to find the item and in the worst-case scenario the item is at the end of the list hence `len(the-list)` TU. In this case the list has only 2 values so we can consider it 1 TU. `n = n//3` is 1 TU. So the whole while loop takes `3 * x` TU with x is how many times the loop repeats. Since we're converting from base 10 to base 3, the loop ends after roughly $log_3(n)$ because we're dividing n by 3 repeatedly until it becomes 0 here. The same goes with binary conversion, i.e. $log2(n)$ times.

In summary, in the worst-case scenario, it's going to take some $3 * log3(n)$ TU. In big O notation, multiplying $log_3(n)$ by a constant is the same as $log_3(n)$ so the time complexity is $\mathcal{O}(log_3(n))$.

Just think of it as *"oh it's taking about $log_3(n)$ computations"*.

---

326. Power of Three

https://leetcode.com/problems/power-of-three/description/

In this problem, we're given an integer $-2^{31} <= n <= 2^{31} - 1$. The question is if it's a power of 3.

Now let's brute-force it. A power of 3 is anything from the list:
$3^0, 3^1, 3^2, 3^3, 3^4, 3^5 \ldots$
or:
$1, 3, 9, 27, 81, 243, 729 \ldots$

One way to brute-force it is to create the whole list, then check if n is in the list. How many elements are there in the list? Let x be the maximum power in the list then $3^x = 2^{31} - 1$ so $x \approx log_3(2^{31} - 1) \approx 10^9$

That's ~ a billion numbers. **The rule of thumb is if your algorithm requires storing a billion values or taking a billion calculations then it's not the way to go.**

We gotta find a better way. And ofc it's right there. A power of 3 has to be divisible by 3. And if we keep dividing it by 3, at one point it has to reach 1.

What are the edge cases? If it's non-positive (<=0) then it cannot be a power of 3. OK let's code:

```
class Solution:
    def isPowerOfThree(self, n):
        if n <= 0:
```

```
            return False
        while True:
            if n == 1:
                return True
            if n % 3 != 0:
                return False
            n //= 3
```

First we check if n is not positive, in which case we return False. Now we need a loop to keep dividing n by 3. A `while True` works, but **remember to make sure the loop would end!** Inside the loop we check if n is 1 which means either n is 1 from the beginner or we've been dividing n by 3 and it's reached 1. In any case, this means n is a power of 3 so return True.

If n is not 1, we keep checking if n is divisible by 3. If not then return False. Why do we check n is 1 before we check if it's divisible by 3? Think about it, if we do it the other way round, 1 is not divisible by 3, in which case the code returns False. But 1 is indeed a power of 3, and it should return True. The order of the logic is very important here!

If n is not 1 but it is divisible by 3, we just divide it by 3. The loop has to end somewhere in the 2 if-conditions. Why? Any number is either divisible by 3 or not. If it is not, then we're out of the loop in the 2nd if-condition. If it is, then when we divide n by 3, we either got 1 - so out of the loop in the next 1st if-condition, or we start again with a new number > 1, in which case the logic applies again.

Now that's one solution. But I want to bring us another solution related to the current topic: bit manipulation. Recall that the powers of 3 are:
$(3^0)_{10} = 1_3$
$(3^1)_{10} = (10)_3$
$(3^2)_{10} = (100)_3$
...
$(3^5)_{10} = (100000)_3$
...

A power of 3, if represented in base 3, is digit 1 followed by any number of 0!!! So if we convert the number to base 3, we can check if it is a power of 3 by checking if the most-significant-bit (MSB) is 1 and the rest is 0.

```
class Solution:
    def isPowerOfThree(self, n):
        if n <= 0:
```

```
            return False
        rep = []
        while n:
            rep.append(n % 3)
            n //= 3
        return rep[-1] == 1 and all([x == 0 for x in rep[:-1]])
```

First, we check if n is not positive in which case it's not a power of 3 and return False. Otherwise, we convert n to base 3. Recall this can be done by repeatedly getting the LSB which is the remainder of division by 3, and perform integer division by 3 to n. We do this until n is 0, in which case we break out of the loop.

Remember we are extract the LSB first & append to the result so in the return, the MSB is at the end of the list and the LSB bit is at the top. To access the MSB at the bottom, we use `rep[-1]` which means the last item on the list.

We check if all digits other than the MSB are 0 by `all([x == 0 for x in rep[:-1]])` . How does this work? First `rep[:-1]` means we're looking at the sublist from the first element to the element before last. Please read this tutorial on list slicing https://www.geeksforgeeks.org/python-list-slicing/.
`[x == 0 for x in rep[:-1]]` means:

- create a new list
- for each element in the list `rep[:-1]` check if it's is 0
- if it is, append True to the new list
- if it isn't, append False to the new list
- return the new list

`all` checks if all the elements in the list is True:

- `all([True, True, True])` is True
- `all([True, False, True])` is False
- ...

Note that we can also drop the `[]` when using all, i.e. `all(x == 0 for x in rep[:-1])` is enough.

In sum, the first component of the return checks if the MSB is 1, the second component checks if the rest of the digits are all 0.

Time complexity

Similarly, most of the computation is done in the while loop and it takes $\mathcal{O}(log_3(n))$.

PLEASE READ THE FOLLOWING TUTORIALS:

and try to use them from now on.

- list slicing: https://www.geeksforgeeks.org/python-list-slicing/
- list comprehension: https://www.geeksforgeeks.org/python-list-comprehension/
- read more on `all` and `any` https://www.geeksforgeeks.org/python-check-if-all-elements-in-list-follow-a-condition/

---

231. Power of Two

https://leetcode.com/problems/power-of-two/description/

This problem can be approached the same way as Power of three above. Try to solve it yourself first. I'll skip the explanation and provide the code below.

```
def isPowerOfTwo(self, n):
    if n <= 0:
        return False
    rep = []
    while n:
        rep.append(n % 2)
        n //= 2

    return rep[-1] == 1 and all([x == 0 for x in rep[:-1]])
```

or:

```
def isPowerOfTwo(self, n):
    if n <= 0:
        return False
    while n:
        if n == 1:
            return True
        if n % 2 != 0:
            return False
        n //= 2
```

Now we'll discuss about another solution. This is new so don't worry if you didn't come up with it from the first place. The technique is called **"drop the lowest set bit"**. In binary representation, 1 is a set bit and 0 is an unset bit. The lowest set bit is the right most bit of 1. For example, the bold 1(s) below are the lowest set bit:

$1_{10} = (\mathbf{1})_2$

$2_{10} = (10)_2$
$3_{10} = (11)_2$
$282_{10} = (100011010)_2$

To drop the lowest bit means to flip it from 1 to 0. And to do so for number `x` :

```
x &= x - 1
```

Recall `&` is a bit-wise AND operator, meaning `a & b` = represent a and b in binary, perform a bit-wise AND operation, return the result.
For a refresher on bit-wise binary operators in Python, please read https://www.geeksforgeeks.org/python-bitwise-operators/

So why does `x &= x -1` drops the lowest set bit of x? Let's try some examples:

- if `x = 1` then `x - 1 = 0` → $(1)_2 \wedge (0)_2 = (0)_2$
- if `x=2` then `x - 1 = 1` → $(10)_2 \wedge (01)_2 = (00)_2$
- if `x=3` then `x - 1 = 2` → $(11)_2 \wedge (10)_2 = (10)_2$
- if `x=282` then `x -1 = 281` → $(100011010)_2 \wedge (100011001)_2 = (100011000)_2$

Try above examples with pen/paper if you want.

Why does it work? Think about the lowest set bit and any number of 0s after it. It could be `1` , or `10` , or `100000` ... When we subtract x by 1, the bits on the left (more significant) of the lowest set bit stay the same, but the lowest set bit and (none or more) 0s on its right become 0 followed by (none or more) 1s. For example:

- `1` → `0` (1 - 1 = 0)
- `10` → `01` (2 - 1 = 1)
- `100` → `011` (4 - 1 = 3)
- ...

Let A be the bits on the left of the lowest set bit, let's work through some example:
just the lowest set bit
Then:
$(x)_{10} = (\{A\}1)_2$
$(x-1)_{10} = (\{A\}0)_2$

Now if you perform bit-wise AND, A should stay the same, but `1^0` becomes 0. So the lowest set bit has been dropped.

lowest set bit followed by some zeros
say 3 zeros, then:
$(x)_{10} = (\{A\}1000)_2$
$(x-1)_{10} = (\{A\}0111)_2$

You can see that if we perform bit-wise AND, again A stays the same, but `1000` and `0111` cancel each other → `0000` .

It all boils down to the fact that if a number in binary is 1 followed by any number of zeros, its subtraction by 1 cancels it out in bit-wise AND operation:
$1 \wedge 0 = 0$
$10 \wedge 01 = 00$
$100 \wedge 011 = 000$
...

!!! In a sum, to drop the lowest set bit, perform `x &= x - 1` !!!

New solution
We can apply this new technique into solving this problem. Recall that a power of 2 are:
$2^0, 2^1, 2^2, 2^3, 2^4, 2^5 \dots$

In binary representation, these numbers are always a 1 followed by some (or none) numbers of 0s. This means if we drop the set bit, there's no digit 1 left in the binary representation → it becomes 0.

```
class Solution:
    def isPowerOfTwo(self, n):
        if n <= 0:
            return False
        n &= (n - 1)
        return n == 0
```

The algorithm first checks if the number is not positive, in which case it can't be a power of 2 so return False. Otherwise, it drops the lowest set bit of n. If the result is 0, this means the n in binary representation is a digit 1 followed by some (or none) numbers of 0, so it is a power of 2. Otherwise, it cannot be a power of 2.

Time complexity
It takes 1 TU to calculate `n-1` ,1 TU to do the AND operation, 1 TU to run the check `if n <= 0` . In sum it takes a constant time so $\mathcal{O}(1)$.

---

342. Power of Four

https://leetcode.com/problems/power-of-four/

Should be similar to power of 3 and power of 2 problems. If you're still confused, try convert the following powers of 4 into base-4:
$4^0, 4^1, 4^2, 4^3, 4^4, 4^5 \dots$
You'll see it's 1 followed by some (or none) number of 0s.

Alternatively, we can think of power of 4 in terms of binary representation. Recall that multiplying a number by 2 is the same as shifting its binary representation to the left by 1 position:
$1_{10} = (1)_2 \rightarrow 2_{10} = (10)_2 \rightarrow 4_{10} = (100)_2$
or
$5_{10} = (101)_2 \rightarrow 10_{10} = (1010)_2 \rightarrow 20_{10} = (10100)_2$

Multiplying by 4 equals to multiplying by 2 then multiplying 2, meaning shifting to the **left by 2 positions**:
$5_{10} = (101)_2 \rightarrow 20_{10} = (10100)_2 \rightarrow 80_{10} = (1010000)_2$

We can create the list of powers of 4 starting from 1, then multiply the previous by 4. This means a power of 4 in binary representation should be a digit 1 followed by an even number of digit 0. So another solution for this problem is to convert the number into binary, then check if it starts with 1, and there's only one digit 1, and an even number of digits 0.

```
class Solution:
    def isPowerOfFour(self, n):
        if n <= 0:
            return False
        rep = []
        while n:
            rep.append(n % 2)
            n //= 2

        num0 = num1 = 0
        for x in rep:
            if x == 0:
                num0 += 1
            else:
                num1 += 1
        return rep[-1] == 1 and num1 == 1 and num0 % 2 == 0
```

Let k be the length of `rep` in above code, k is essentially the length of binary representation of n. We can see that the number of computation is about `2 * k` . Most of the numbers we deal with are either 32-bit or 64-bit so k is a constant. Indeed, the problem's constraint states $-2^{31} \leq n \leq 2^{31} - 1$ so it's only 32-bit. This means the algorithm runs in a constant time → $\mathcal{O}(1)$.

2595. Number of Even and Odd Bits

https://leetcode.com/problems/number-of-even-and-odd-bits/

This problem should be easy enough. Just read the description carefully & apply binary conversion.

```python
class Solution:
    def evenOddBit(self, n):
        rep = []
        while n:
            rep.append(n % 2)
            n //= 2

        ret = [0, 0]
        for idx, val in enumerate(rep):
            if val == 1:
                if idx % 2 == 0:
                    ret[0] += 1
                else:
                    ret[1] += 1
        return ret
```

---

504. Base 7

https://leetcode.com/problems/base-7/

One thing I think this problem should make clearer is for negative numbers. If the number if negative, just convert its positive counterpart into base 7 then append `-` at the front.

The solution is straight forward, we convert the number from base 10 to base 7, return it. The major point here is the problem expects the a string not a list as we've been doing. We'll take this chance to revisit **integer to string conversion & string joining**. Let's jump straight into the solution:

```
class Solution:
    def convertToBase7(self, num):
        if num == 0:
            return "0"
        ret = []
        is_neg = num < 0
        num = abs(num)
        while num:
            ret.append(num % 7)
            num //= 7
        ret = [str(x) for x in ret][::-1]
        ret = ''.join(ret)
        return ret if not is_neg else '-' + ret
```

First we clear the edge case of 0.

Next, we use a list to keep the digits when converting from base 10 to base 7. If the number of negative, we use its positive counterpart in the conversion and have a flag `is_neg` to check later on if we should append `-` at the front of the result or not.

The conversion is straight-forward, keep taking the remainder of division by 7 then perform integer division until the number becomes 0. Recall that we are converting by calculating the LSB bits first, so `ret` contains the representation in reverse. To get it right, we can simply reverse the list by using slicing technique:

```
ret = ret[::-1]
```

In addition, all the elements in `ret` are integers, but we want string, so we convert them all into string by calling `str(x)`. We do this for all elements by using list comprehension: `ret = [str(x) for x in ret]`. In the code, we merge the two operations (reverse + convert to string) into one:

```
ret = [str(x) for x in ret][::-1]
```

Python can merge a list of strings using the `join` method: `ret = ''.join(ret)`. This says I'm going to merge all the strings in ret together, separated by `''` which is an empty space. So if `ret = ["0", "1", "2"]` then `''.join(ret)` is `012`.

Finally, we check if originally the number is negative or not. If not we return the representation, else we append `-` before returning.

Time complexity:
The base conversion takes ~ $log_7(num)$ TU. The representation also has $log_7(num)$ elements. Reversing a list of n values takes n TU, same as string conversing for each element of the list. So the reversing and string conversion in list comprehension each takes around $log_7(num)$ TU. Overall, it's a constant time $log_7(num)$ hence the time complexity is $\mathcal{O}(log_7(num))$.

Revisit:

- https://www.geeksforgeeks.org/python-list-slicing/ if you're confused about `ret[::-1]`
- more on reversing a list https://www.geeksforgeeks.org/python-reversing-list/
- https://www.geeksforgeeks.org/python-list-comprehension/ if you're still confused about list comprehension
- More on `join` method https://www.geeksforgeeks.org/python-string-join-method/

---

461. Hamming Distance

https://leetcode.com/problems/hamming-distance/

Read the description carefully. In essential, if we take 2 numbers, convert them into binary, then the hamming distance of the 2 numbers is the number of corresponding bits in the 2 representations that differ in values.

The solution is straight-forward. We convert x and y into binary at the same time. At each time step, we check if the bits are the same or different. If different, we increase the hamming distance.

```
class Solution:
    def hammingDistance(self, x, y):
        ret = 0
        while x or y:
            bit_x = x % 2
            bit_y = y % 2
            x //= 2
            y //= 2
            ret += 1 if bit_x != bit_y else 0
        return ret
```

Recall that in base conversion, we stop when the number becomes 0 because continuing to do so just returns more 0 while the number itself doesn't change in value (already 0 so divided by the base still is 0). In our case, we don't know if x becomes 0 first or y becomes 0 first, or both become 0 at the same time (well, we can, we just don't care). It doesn't matter which reaches 0 first actually because it's still a valid binary representation, i.e. $(101)_2 = (000000101)_2$ and it shouldn't change the hamming distance. However, we should stop if both numbers have become 0, their next bits are always equally 0 hence not affecting the hamming distance anymore.

Time complexity

Converting a number n to binary takes $\mathcal{O}(log_2(n))$. In this case the time complexity should be $\mathcal{O}(log_2(max(x, y)))$ (I'll leave it to you to work out why).

---

190. Reverse Bits

https://leetcode.com/problems/reverse-bits/

The problem has 2 constraints:

- the number if unsigned, i.e. you don't have to worry about negative number and 2-complement
- the number is 32-bit

A simple brute-force solution is to calculate the binary representation of the number, reverse it, then calculate the new number from the new representation.

```
class Solution:
    def reverseBits(self, n):
        rep = [0] * 32
        idx = 0
        while n:
            rep[idx] = n % 2
            idx += 1
            n //= 2
        rep = rep[::-1]
        ret = 0
        power = 1
        for x in rep:
            ret += power * x
            power *= 2
        return ret
```

We know the number is 32-bit, so we prepare ahead a list of 32 values. The default value of all elements is 0 and during binary conversion, if a bit is 1 we simply change it as such. After obtaining the representation, we reverse it with list slicing, then simply recalculate the value using the equation:

$$m = 2^0 * a_0 + 2^1 * a_1 + 2^2 * a_2 + \ldots + 2^n * a_n$$

Now let's a good solution, but we can do something more fun. Think of reversing, it's basically swapping isn't it? If we have a list of 2 elements, reversing = swapping them. With 3 elements, reversing = swapping the 0-th and the 2-th. With 4 elements, reversing = swapping 0-th with 3-th, 1-th with 2-th. **In general, if for a list of size n, reversing means swapping element i^{th} with element $(n-1-i)^{th}$ (note I'm using base-0 indexing here as in Python).**

Our number is 32-bit, meaning we want to swap 0-th with 31-th, 1-th with 30-th...15-th with 16-th, and stop here (right?). Now say we're swapping 0-th and 31-th. If they're both 0 or both 1, we don't need to do anything. If they're 0 and 1, then we swap. If it's a list, then swapping is simple:

```
rep[0], rep[31] = rep[31], rep[0]
```

But this requires representing the number into a list of 0/1. Let's say we want to avoid that, how can we do it? Hint: XOR is your friend.

Recall XOR truth table:

IN-1	IN-2	OUT
0	0	0
0	1	1
1	0	1
1	1	0

For any bit x (0 or 1), x ^ 0 = x, but x ^ 1 = ~x. So XOR-ing with 0 keeps the value, XOR-ing with 1 flips the value. We can use this fact to ***swap any two bits of different values***. Say the two bits are at position i-th and j-th. We can build a mask of the same size as the input number, all of values 0 everywhere except for positions i-th and j-th where it's 1. If we bit-wise XOR the input number of this mask, all positions except i-th and j-th stay the same because we're XOR-ing them with 0. For the bits at i-th and j-th, they are of different values as we stated before, assumingly 0 and 1. Since we're XOR-ing them with 1, they become 1 and 0. So essentially we've swapped their values!

```
class Solution:
    def reverseBits(self, n):
        def swap_bits(x, i, j):
```

```
            if (x >> i) & 1 != (x >> j) & 1:
                bit_mask = (1 << i) | (1 << j)
                x ^= bit_mask
            return x

        for i in range(0, 16):
            n = swap_bits(n, i, 31 - i)
        return n
```

A few things are going on here. First we try to grab the bit at i-th and j-th. We can get the i-th bit position by: `(x >> i) & 1`. The shifting to the right by i steps moves the i-th bit to the LSB position. The bitwise AND operation with 1 making all bits except the LSB 0. Let's do an example, getting the 4-th (i.e. the fifth bit, we're doing base-0 indexing here like Python) bit of number 432. 432 in binary is (110110000).

- Shifting by 4 → 11011
- 1 in binary is 00001
- bitwise AND → 00001

Now we can see how the code extracts i-th and j-th bits. Next we check if they're different. If yes, we build the mask. The mask for i-th should be all 0 except 1 at i-th. Same for mask for j-th. Since $1 \vee 0 = 1$ and $1 \vee 1 = 1$, combining the two masks is simply bit-wise OR them.

Time complexity: Both algorithms take about `32C` TU with `C` being some constant, so the time complexity is $\mathcal{O}(1)$.

1720. Decoded XORed Array

https://leetcode.com/problems/decode-xored-array/

This problem reinforces a simple but effective XOR trick. But before we delve into that, let's look into a brute-force solution.

Given an array `arr` with n integers, we encode it into an array `encoded` with length n-1 in which `encoded[i] = arr[i] ^ arr[i+1]`. Now we're given the array `encoded` instead and the first value in `arr`, the problem asks us to fill in the rest of n-1 integers in `arr`.

We find the rest of the values 1 at a time. Starting off, we need to find `arr[1]` given `arr[0]` and `encoded[0]`. We know that:

```
encoded[0] = arr[0] ^ arr[1]
```

A really dumb brute-force solution is to try all values for `arr[1]` and check if `arr[0] ^ arr[1]` yields `encoded[0]`. That'll take forever as there are infinite integer numbers. A better brute-force is to look at binary representation. We find `arr[1]` 1 bit at a time. Let's look into the XOR truth table:

arr[0]	arr[1]	encoded[0]
0	0	0
0	1	1
1	0	1
1	1	0

This means we can look at the corresponding bits in `arr[0]` and `encoded[0]`, and if they are:

- (0, 0) then the corresponding bit in `arr[1]` is 0
- (0, 1) then the corresponding bit in `arr[1]` is 1
- (1, 1) then the corresponding bit in `arr[1]` is 0
- (1, 0) then the corresponding bit in `arr[1]` is 1

So basically if the bits in `arr[0]` and `encoded[0]` are the same then the corresponding bit in `arr[1]` is 0 else it's 1.

```
class Solution:
    def decode(self, encoded, first):
        ret = [first]
        for i, xor in enumerate(encoded):
            arr0 = ret[i]
            arr1 = 0
            power = 2**0
            while arr0 or xor:
                arr0_bit = arr0 % 2
                xor_bit = xor % 2
                arr0 //= 2
                xor //= 2
                arr1_bit = 1 if arr0_bit != xor_bit else 0
                arr1 += power * arr1_bit
                power *= 2
            ret.append(arr1)
        return ret
```

This solution works, but we can do better. Notice how when we decode `arr[1]` given the values of `arr[0]` and `encoded`, it looks just like another XOR table? because it is. See the math here:

```
a ^ b = c
a ^ (a ^ b) = a ^ c
(a ^ a) ^ b = a ^ c
0 ^ b = a ^ c
b = a ^ c
```

So to figure out `b` given `a,c` we can just do `b = a ^ c` !!! This brings about a much simpler solution :)

```python
class Solution:
    def decode(self, encoded, first):
        ret = [first]
        for i, xor in enumerate(encoded):
            ret.append(ret[i] ^ xor)
        return ret
```

We can consider one XOR operation takes 1 TU then the time complexity is $\mathcal{O}(n)$. We need an array of n values to hold the result so the space complexity is also $\mathcal{O}(n)$.

136. Single Number

https://leetcode.com/problems/single-number/description/

This is a very classic problem using bit manipulation. It is medium but it shouldn't be. Hint: XOR.

> **Any number XOR itself is 0** -- said some wise man ---

Since each element in the array appears twice except for one number. If we XOR all elements in the array, the duplicates should cancel themselves leaving the number to be found.

```python
class Solution:
    def singleNumber(self, nums):
        # THINK ABOUT WHY IT SHOULD START FROM 0 INSTEAD OF 1
        ret = 0
        for n in nums:
            ret ^= n
        return ret
```

389. Find the Difference

https://leetcode.com/problems/find-the-difference/description/

This is the same as above problem except with strings instead of integers and it's about time to introduce `ord()` and `chr()` .
This is a good introduction on the two functions https://www.digitalocean.com/community/tutorials/python-ord-chr

If you don't want to dive into Unicode codepoint, just think of each letter has a position in Python. Given a character, `ord()` gives you its position, and given a position `chr()` gives you the character.

```
In [1]: ord("a")
Out[1]: 97

In [2]: ord("b")
Out[2]: 98

In [3]: ord("c")
Out[3]: 99

In [4]: ord("A")
Out[4]: 65

In [5]: ord("B")
Out[5]: 66

In [6]: ord("Z")
Out[6]: 90
```

As you can probably guess, the letters in alphabet have their positions sorted so `ord("A") < ord("Z")` for example.
Using `chr()` , we can quickly find out which character it is instead of having to manually consult the alphabet:

```
In [7]: chr(97)
Out[7]: 'a'

In [8]: chr(ord("a") + 10)
Out[8]: 'k'

In [9]: chr(ord("a") + 25)
Out[9]: 'z'
```

In this problem, the 2 strings `s` and `t` have the same characters but in different order and `t` has 1 more character. If we XOR all the characters in them, we should get that extra character in `t` . We can't XOR on strings, but we can XOR on their `ord()` .

```
class Solution:
    def findTheDifference(self, s, t):
        ret = 0
        for ch in s:
            ret ^= ord(ch)
        for ch in t:
            ret ^= ord(ch)
        return chr(ret)
```

268. Missing Number

https://leetcode.com/problems/missing-number/description/

This problem should be the same the two problems above :)

231. Power of Two

Given an integer, write a program to determine if it is a power of 2 or not.

What's a power of 2?

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

...

Looking from the decimal perspective, it's hard to see a pattern here (1, 2, 4, 8...). But recall from binary representation, multiplication by 2 is essentially shifting to the left 1 position:

$2^0 = 1 = 1_2$

$2^1 = 2 = 10_2$

$2^2 = 4 = 100_2$

$2^3 = 8 = 1000_2$

...

The pattern here is a power of 2, in binary representation, is always a 1 followed by zero or more 0s.

If you haven't worked out the above strategy, I want you to stop reading here and try to implement it and check if it passes all the tests.

The implementation is simple:

- convert number into binary representation
- check if the left-most bit is 1
- check the rest of the bits are 0

Let me introduce a better trick for this problem: **drop right-most set bit!**
The right-most set bit is the right-most 1 in the binary representation of a number. To drop it means to flip it to 0. How do we do it fast?

```
x = x & (x - 1)
```

Let's see a few examples:

x	1**100**	10011**1**	1010101010
x - 1	1**011**	10010	1010101001
x & (x - 1)	1**000**	10010	1010101000

Try to come up with a few examples yourself.
How can this trick be used for this problem? If a number is a power of 1, its binary representation is 1 followed by zero or more 0s. So that only 1 is the right-most set bit, and if we drop it, the number should become 0!

```
class Solution:
    def isPowerOfTwo(self, n):
        if n <= 0:
            return False
        n &= (n - 1)
        return n == 0
```

461. Hamming Distance

https://leetcode.com/problems/hamming-distance/
Hamming distance between 2 integers is the number of positions in their binary representation where the bits are different. A straight-forward approach is to convert both into binary and check. Since we only need to check the bits at the same position, we can do the check at the same time as the binary conversion.

```
while a or b:
    # we stop binary conversion when number is 0.
```

```
    # but a and b can become 0 at different time.
    # but then think about it, it doesn't matter if a is 0 before b or
vice versa.
    # assuming a becomes 0 before b, its binary representation after it
becomes 0 is just 0s.
    # the correct time to stop is when both a and b are 0s because after
that there's no change to the hamming distance
    # hence the condition (a or b) which is False only if a and b are both
0s
    a_bit = a % 2
    b_bit = b % 2
    dist += 1 if a_bit != b_bit else 0
    a //= 2
    b //= 2
```

78. Subsets

https://leetcode.com/problems/subsets/

Given a list of numbers, we are asked to return all of its subsets, i.e. it's power set. The power set contains all possible subsets of a set including the empty set. It's simple to see an example, suppose our set is {A, B, C}, then all possible sets that can be created from its elements are:

```
{A}, {B}, {C}
{A, B}, {B, C}, {A, C}
{A, B, C} # yes a set is its own subset too
```

Throw in the empty set `{}` then we have the power set. Note for set, we don't care about order, so `{A, B}` and `{B, A}` are the same.

How can we generate all of them? Think about the process of generating a subset: We go through all elements in the set, we ask the question: should we put this number in the subset? For example `{A}` means we go to A, choose to pick A, but when we see B and C, we decide not to pick either of them.

If we consider picking as 1 and not picking as 0, then we start to see a pattern emerges:

A	B	C	Subset
0	0	0	{}
1	0	0	{A}
0	1	0	{B}
0	0	1	{C}
1	1	0	{A, B}

0	1	1	{B, C}
1	0	1	{A, C}
1	1	1	{A, B, C}

It looks familiar, let's re-arrange the table:

A	B	C	Subset
0	0	0	{}
0	0	1	{C}
0	1	0	{B}
0	1	1	{B, C}
1	0	0	{A}
1	0	1	{A, C}
1	1	0	{A, B}
1	1	1	{A, B, C}

These are binary representations of all the numbers from 0 → 2^3 - 1.

If you try with a larger set: 4 elements, 5 elements, ... you should see the same pattern too. This brings us to our algorithm:

- n = number of elements in set
- for all number from 0 to 2^n - 1
  - generate the number's binary representation
  - if a bit is 1, pick its corresponding element in original set to the subset

```
class Solution:
    def subsets(self, nums):
        res = []
        for x in range(2 ** len(nums)):
            subset = []
            idx = 0
            while x:
                if x % 2:
                    subset.append(nums[idx])
                x //= 2
                idx += 1
            res.append(subset)
        return res
```

DIY

https://leetcode.com/problems/binary-number-with-alternating-bits/

https://leetcode.com/problems/binary-gap/description/

https://leetcode.com/problems/prime-number-of-set-bits-in-binary-representation

https://leetcode.com/problems/counting-bits/description/

https://leetcode.com/problems/find-the-original-array-of-prefix-xor

https://leetcode.com/problems/decode-xored-permutation/description/

https://leetcode.com/problems/reverse-bits/

https://leetcode.com/problems/minimum-bit-flips-to-convert-number/

https://leetcode.com/problems/sum-of-digits-in-base-k/ → max base is 10 so we don't have to worry about using letters

https://leetcode.com/problems/minimum-bit-flips-to-convert-number/description/