Bit Manipulation

Decimal Based System:

Decimal is a term that describes the base-10 **number system**, probably the most commonly used **number system**. The **decimal number system** consists of ten single-digit **numbers**: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9

But Computer only understands only o's and 1's ➔ Binary Based System

Binary Based System:

Binary number system, in mathematics, positional numeral system employing **2 as the base** and so requiring only **two different symbols** for its **digits, 0 and 1**, instead of the usual 10 different symbols needed in the decimal system.

So,

0	0
1	1
2	10

3	11
4	100
5	101

6	110
7	111
8	1000

9	1001

Conversion of Decimal Number to Binary Number:

In decimal system, a number 171 can be written as

171 = $1 * 10^2 + 7 * 10^1 + 1 * 10^0$

2561 = $2 * 10^3 + 5 * 10^2 + 6 * 10^1 + 1 * 10^0$

In the same way, the binary system has only 2 digits, (0, 1). So,

Binary to Decimal Conversion:

Ex: 111 = $1 * 2^2 + 1 * 2^1 + 1 * 2^0$

= $4 + 2 + 1 = \underline{7}$

1000 = $1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$

= $8 + 0 + 0 + 0 = \underline{8}$

1011 = $1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$

= $8 + 0 + 0 + 0 = \underline{8}$

Q) Given a string in binary, convert to decimal integer.

Ex: “100” → 4

Decimal to Binary Conversion:

Decimal No = 55

2	55
1	27
1	13
1	6
0	3
1	1

55 = 110111

Decimal No = 7

2	7
1	3
1	1

7 = 111

Decimal No = 28

2	28
0	14
0	7
1	3
1	1

28 = 11000

The program would be,

```
def convertDecimaltoBinary(num):
    binary_string = ""
    while num > 0:
        rem = num % 2
        binary_string += str(rem)
        num //= 2

    return binary_string[::-1]

if __name__ == "__main__":
    print(convertDecimaltoBinary(155))
```

OUTPUT:

10011011

GATES

We have three types of Gates, ***AND, OR, XOR***

Truth Table for AND

x	y	x & y
1	1	1
0	1	0
1	0	0
0	0	0

Truth Table for OR

x	y	x | y
1	1	1
0	1	1
1	0	1
0	0	0

Truth Table for XOR

x	y	x ^ y
1	1	0
0	0	0
1	0	1
0	1	1

If anyone is zero, then x & y = 0
If anyone is one, then x | y = 0
If both are same, x ^ y = 0 and if both are different then x ^ y = 1

Bitwise Operators:

The **bitwise shift** operators are the **right-shift** operator (>>), which moves the bits of **shift**-expression to the **right**, and the left-**shift** operator (<<), which moves the bits of **shift**-expression to the left.

- & (bitwise AND)
- | (bitwise OR)
- ^ (bitwise XOR)
- << (bitwise left shift)
- >> (bitwise right shift

The & Operator

The & operator compares each binary digit of two integers and returns a new integer, with a 1 wherever both numbers had a 1 and a 0 anywhere else. A diagram is worth a thousand words, so here's one to clear things up. It represents doing `37 & 23`, which equals `5`.

	Binary Digits							
37	0	0	1	0	0	1	0	1
& 23	0	0	0	1	0	1	1	1
= 5	0	0	0	0	0	1	0	1
Digits Were Both 1?	No	No	No	No	No	Yes	No	Yes

Notice how each binary digit of 37 and 23 are compared, and the result has a 1 wherever both 37 and 23 had a 1, and the result has a 0 otherwise.

When we compare two booleans, we normally do `boolean1 && boolean2`. That expression is only true if both `boolean1` and `boolean2` are true. In the same way, `integer1 & integer2` is equivalent, as the & operator only outputs a 1 when both binary digits of our two integers are 1.

Here's a table that represents that idea:

& Operator Truth Table		
Digit 1:	Digit 2:	Result:
0	0	0
0	1	0
1	0	0
1	1	1

The | Operator

Up next is the bitwise OR operator, |. As you may have guessed, the | operator is to the || operator as the & operator is to the && operator. The | operator compares each binary digit across two integers and gives back a 1 if *either* of them are 1. Again, this is similar to the || operator with booleans.

| Operator Truth Table		
Digit 1:	Digit 2:	Result:
0	0	0
0	1	1
1	0	1
1	1	1

Let's take a look at the same example as before, except now using the | operator instead of the & operator. We're now doing `37 | 23` which equals 55:

	Binary Digits							
37	0	0	1	0	0	1	0	1
| 23	0	0	0	1	0	1	1	1
= 55	0	0	1	1	0	1	1	1
Either Digit is 1?	No	No	Yes	Yes	No	Yes	Yes	Yes

The ^ Operator

We're back to the bitwise operators, and up next is the bitwise XOR operator. There is no equivalent boolean operator to this one.

The ^ operator is similar to the & and | operators in that it takes an `int` or `uint` on both sides. When it is calculating the resulting number, it again compares the binary digits of these numbers. If one or the other is a 1, it will insert a 1 in to the result, otherwise it will insert a 0. This is where the name XOR, or "exclusive or" comes from.

^ Operator Truth Table		
Digit 1:	**Digit 2:**	**Result:**
0	0	0
0	1	1
1	0	1
1	1	0

Let's take a look at our usual example:

	Binary Digits							
37	0	0	1	0	0	1	0	1
^ 23	0	0	0	1	0	1	1	1
= 50	0	0	1	1	0	0	1	0
Only one digit is 1?	No	No	Yes	Yes	No	No	Yes	No

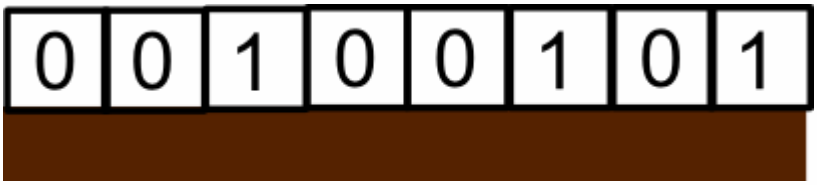
The ^ operator does have uses - it's especially good for toggling binary digits - but we won't cover any practical applications in this article.

The << Operator

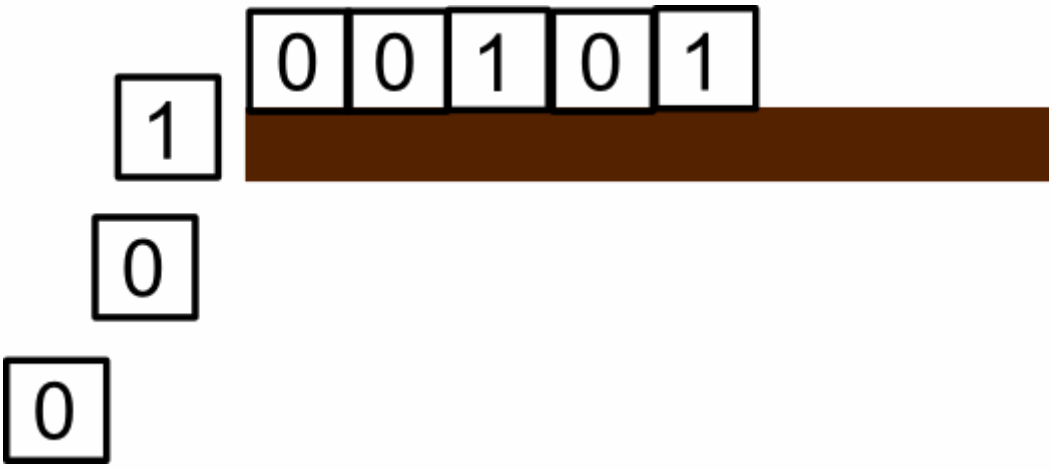
We're now on the bitshift operators, specifically the bitwise left shift operator here.

These work a little differently than before. Instead of comparing two integers like &, |, and ^ did, these operators shift an integer. On the left side of the operator is the integer that is being shifted, and on the right is how much to shift by. So, for example, 37 << 3 is shifting the number 37 to the left by 3 places. Of course, we're working with the binary representation of 37.

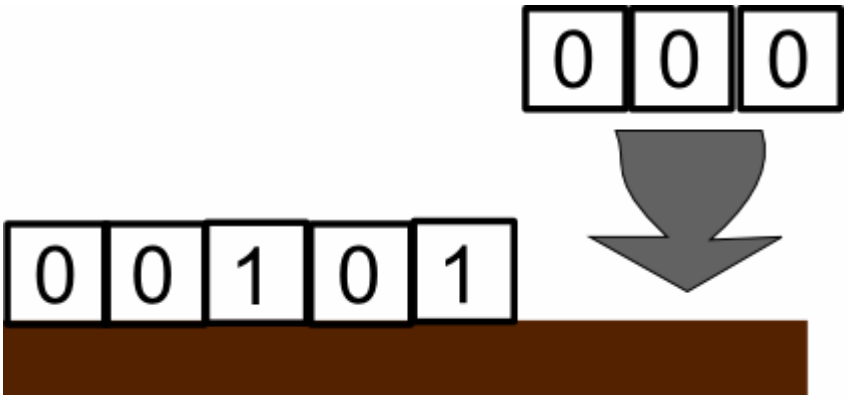
Let's take a look at this example (remember, we're just going to pretend integers only have 8 bits instead of 32). Here we have the number 37 sitting on its nice block of memory 8 bits wide.



Alright, let's slide all the digits over to the left by 3, as `37 << 3` would do:



But now we have a small problem - what do we do with the 3 open bits of memory where we moved the digits from?



Of course! Any empty spots are just replaced with 0s. We end up with 00101000. And that's all there is to the left bitshift.

An interesting feature of the left bitshift is that it is the same as multiplying a number by two to the shiftAmount-th power. So, `37 << 3 == 37 * Math.pow(2,3) == 37 * 8`. If you can use the left shift instead of `Math.pow`, you'll see a huge performance increase.

*You may have noticed that the binary number we ended up with did not equal 37 * 8. This is just from our use of only 8 bits of memory for integers; if you try it in ActionScript, you'll get the correct result. Or, try it with the demo at the top of the page!*

The >> Operator

Now that we understand the left bitshift, the next one, the right bitshift, will be easy. Everything slides to the right the amount we specify. The only slight difference is what the empty bits get filled with.

If we're starting with a negative number (a binary number where the leftmost bit is a 1), all the empty spaces are filled with a 1. If we're starting with a positive number (where the leftmost bit, or most significant bit, is a 0), then all the empty spaces are filled with a 0. Again, this all goes back to two's complement.

While this sounds complicated, it basically just preserves the sign of the number we start with. So `-8 >> 2 == -2` while `8 >> 2 == 2`. I'd recommend trying those out on paper yourself.

Since >> is the opposite of <<, it's not surprising that shifting a number to the right is the same as dividing it by 2 to the power of shiftAmount. You may have noticed this from the example above. Again, if you can use this to avoid calling `Math.pow`, you'll get a significant performance boost.

The other Bitwise Operators are:

- ~ (bitwise NOT)
- >>> (bitwise unsigned right shift)
- &= (bitwise AND assignment)
- |= (bitwise OR assignment)
- ^= (bitwise XOR assignment)
- <<= (bitwise left shift and assignment)
- >>= (bitwise right shift and assignment)
- >>>= (bitwise unsigned right shift and assignment)

https://code.tutsplus.com/articles/understanding-bitwise-operators--active-11301#:~:text=Bitwise%20operators%20are%20operators%20(just,use%20and%20also%20quite%20useful!

NOTE: *We enter decimal number and computer will convert the decimal number to binary number and then shift the binary number and will again convert the binary number to decimal number and give us the answer in decimal number.*

Q) Every number is repeated twice except 3, 2 2 3 4 4 5 5 7 7; find the non-repeated number.

Ex: A = [2 2 1 1 3 4 4 5 5 7 7]

```
m = dict()
for x in a:
        if x not in m;
                m[x] = 1
        else:
                m[x] += 1

for key in m.keys():
        if m[key] == 1:
                return key
```

The time complexity for this program will be O (n) and the space complexity as well would-be O (n).

Can you think of sol^n so that space complexity will be O (1)?

2 2 1 1 3 4 4 5 5 7 7

2 ^ 2 ^ 1 ^ 1 ^ 3 ^ 4 ^ 4 ^ 5 ^ 5 ^ 7 ^ 7

2

- Any number ^ with ‘0’ will give the number itself.
  - X ^ ‘0’ = X
- This is only for ‘0’ and not for other numbers.

The program for using XOR would be:

```
def solve (A):
    ans = A[0]
    for i in range (1, len(A)]:
        ans = ans ^ A[i]
        return ans
```

MAP:

Definition and Usage

The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

Syntax

map(*function*, *iterables*)

Parameter Values

Parameter	Description
function	Required. The function to execute for each item
iterable	Required. A sequence, collection or an iterator object. You can send as many iterables as you like, just make sure the function has one parameter for each iterable.

Examples

Ex: 1

Make new fruits by sending two iterable objects into the function:

```
def myfunc(a, b):
  return a + b

x = map(myfunc, ('apple', 'banana', 'cherry'), ('orange', 'lemon', 'pineapple'))
```

Ex: 2

map(str, [1, 2, 3, 4, 5])

⇨ [“1”, “2”, “3”, “4”, “5”]

Ex: 3

```
def add5[x]:
        return x + 5

map (add5, [1, 2, 3, 4])
```

⇨ [6, 7, 8, 9]

- Map works with lists

https://www.w3schools.com/python/ref_func_map.asp#:~:text=The%20map()%20function%20executes,the%20function%20as%20a%20parameter.

Resources:

Binary Search

https://www.geeksforgeeks.org/binary-search/

https://medium.com/swlh/binary-search-find-upper-and-lower-bound-3f07867d81fb