




# Welcome to the CoGrammar

## Number Bases, Binary Logic and Bitwise Operations (Compressions)

The session will start shortly...

Questions? Drop them in the chat. We'll have dedicated moderators answering questions.



## Coding Interview Workshop Housekeeping

---

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.  
**(Fundamental British Values: Mutual Respect and Tolerance)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: [Questions](#)

## Coding Interview Workshop Housekeeping cont.

---

- For all **non-academic questions**, please submit a query: [www.hyperiondev.com/support](https://www.hyperiondev.com/support)
- Report a **safeguarding** incident: [www.hyperiondev.com/safeguardreporting](https://www.hyperiondev.com/safeguardreporting)
- We would love your **feedback** on lectures: [Feedback on Lectures](#)

# Skills Bootcamp

## 8-Week Progression Overview

### Fulfil 4 Criteria to Graduation

#### ✓ Criterion 1: Initial Requirements

Timeframe: First 2 Weeks  
Guided Learning Hours (GLH):  
Minimum of 15 hours  
Task Completion: First four tasks

**Due Date: 24 March 2024**

#### ✓ Criterion 2: Mid-Course Progress

**60** Guided Learning Hours

Data Science - **13 tasks**  
Software Engineering - **13 tasks**  
Web Development - **13 tasks**

**Due Date: 28 April 2024**

# Skills Bootcamp Progression Overview

## ✓ Criterion 3: Course Progress

Completion: All mandatory tasks,  
including Build Your Brand and  
resubmissions by study period end  
Interview Invitation: Within 4 weeks  
post-course  
Guided Learning Hours: Minimum of  
112 hours by support end date  
(10.5 hours average, each week)

## ✓ Criterion 4: Demonstrating Employability

Final Job or Apprenticeship  
Outcome: Document within 12  
weeks post-graduation  
Relevance: Progression to  
employment or related  
opportunity

**SKILLS  
FOR LIFE**

**SKILLS BOOTCAMPS**



Department  
for Education

# CoGrammar

## Number Bases, Binary Logic and Bitwise Operations (Compressions)

April 2024

# Portfolio Assignment Reviews

Submit your solutions here!





# Learning Objectives

- ❖ **Convert** numbers between different **number bases** (binary, decimal, hexadecimal) and understand their **applications in computer science**.
- ❖ Utilize **binary logic** (AND, OR, NOT, XOR) in Python and JavaScript for **manipulating binary data** and implementing **logic gates**.
- ❖ Perform **bitwise operations** (bitwise AND, OR, NOT, XOR, left shift, right shift) in Python and JavaScript to **manipulate individual bits of data** efficiently.



# Learning Objectives

- ❖ Apply **bitwise operations** for tasks such as **data compression, encryption, and error detection** algorithms.
- ❖ Understand the concepts and applications of **data compression techniques**, distinguishing between **lossy and lossless compression**.

# Number Bases

(not the Roman kind)

I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000






**What do you get when you convert the binary number 10011001 to decimal?**

- A. 150
- B. 153
- C. 154
- D. 158



What do you get when you convert the binary number 10011001 to decimal?

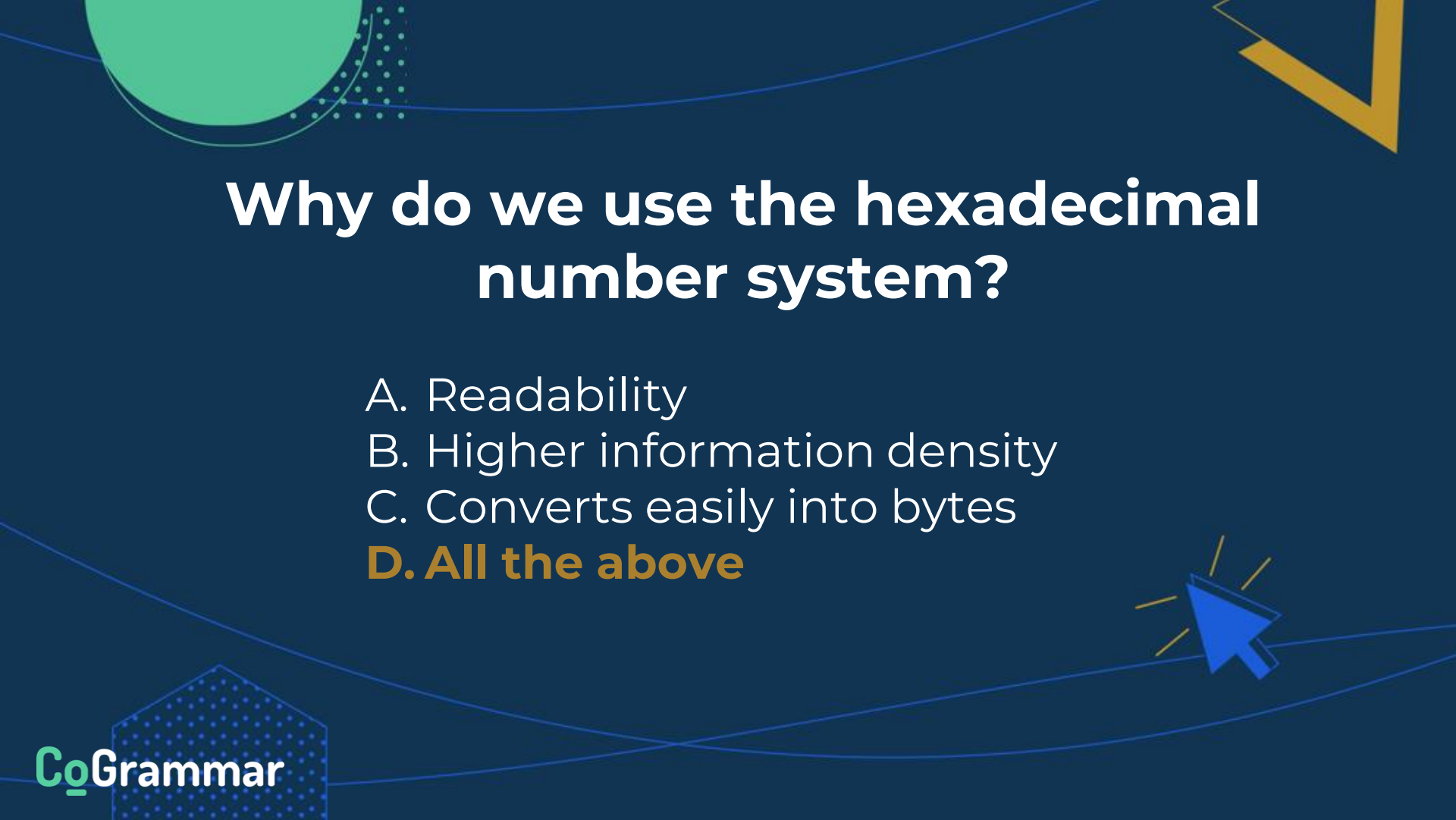
- A. 150
- B. 153**
- C. 154
- D. 158



# Why do we use the hexadecimal number system?

- A. Readability
- B. Higher information density
- C. Converts easily into bytes
- D. All the above





# Why do we use the hexadecimal number system?

- A. Readability
- B. Higher information density
- C. Converts easily into bytes
- D. All the above**

# Types of Number Bases

- ❖ **Decimal number system** (base or radix = **10**)
  - Digits 0 through 9, each place value represents a power of 10.
- ❖ **Binary number system** (base or radix = **2**)
  - Digits 0 and 1, each place value represents a power of 2.
- ❖ **Hexadecimal number system** (base or radix = **16**)
  - Digits 0 through 9 and letters A through F (for values 10 through 15), each place value represents a power of 16.
- ❖ **Octal number system** (base or radix = **8**)
  - Digits 0 through 7, each place value represents a power of 8.



# Types of Number Bases

- ❖ **Decimal number base: 0 – 9** are 10 counting digits, then combinations of two digits, 10 – 99, then combinations of three digits, 100 – 999, and so on.

For example:  $231 = 2 \times 100 + 3 \times 10 + 1 \times 1 = 2 \times 10^2 + 3 \times 10^1 + 1 \times 10^0$

The above combination is true for any number base. Difference is how many digits you have before you move to the next combination.

- ❖ **Binary number system: 0 and 1** are counting digits, binary digit (bit)
  - Start with 0, 1.
  - Two-digit combinations, 10, 11.
  - Three-digit combinations, 100, 101, 110, 111.
  - Four-digit combinations, 1000, .... 1111.

# Types of Number Bases

- ❖ **Octal number system: 0 – 7** are the eight counting digits.
  - Start with 0, 1, 2, 3, 4, 5, 6, 7.
  - Two-digit combinations, 10, 11, 12, 13, ..., 17, 20, ..., 27, 30, ..., 77.
  - Three-digit combinations, 100, 101, 102, ..., 107, 110, 111, ..., 777.
  - Four-digit combinations, 1000, ..., 1007, 1010, 1011, ..., 7777.
- ❖ **Hexadecimal number system: 0 – 9, A, B, C, D, E, F** are the 16 counting digits.
  - A – F represents 10 – 15 in decimal.
  - Two-digit combinations, 10, 11, ..., 19, 1A, 1B, ..., 1F, 20, ..., FF
  - Three-digit combinations, 100, 101, ..., FFF.
  - Four-digit combinations, 1000, ..., FFFF.

# Number Base Conversions

## ❖ Binary to Decimal

Binary: a = 11001

$$\begin{aligned}\text{Decimal: } 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ = 16 + 8 + 0 + 0 + 1 = 25\end{aligned}$$

```
b = "11001"
print("Binary " + b + " to decimal:", int(b,2))
#Output: Binary 11001 to decimal: 25
```

```
let b = "11001";
console.log("Binary " + b + " to decimal:", parseInt(b, 2));
//Output: Binary 11001 to decimal: 25
```

# Number Base Conversions

## ❖ Decimal to Binary

2	123	
2	61	1
2	30	1
2	15	0
2	7	1
2	3	1
	1	1

↑

→

```
decNum = 123
#decNum = int(input("Enter any Decimal Number: "))

def decimalToBinary(n):
    return bin(n).replace("0b", "")

print('Decimal ' + str(decNum) + ' to binary:', decimalToBinary(decNum))
#Output: Decimal 123 to binary: 1111011
```

```
let decNum = 123
function decimalToBinary(n) {
    return (n >>> 0).toString(2);
}
console.log('Decimal ' + decNum + ' to binary:', decimalToBinary(decNum));
//Output: Decimal 123 to binary: 1111011
```

# Number Base Conversions

## ❖ Octal to Decimal

Octal:  $(1071)_8$

$$\begin{aligned}\text{Decimal: } & 1 \times 8^3 + 0 \times 8^2 + 7 \times 8^1 + 1 \times 8^0 \\ & = 512 + 0 + 56 + 1 = (569)_{10}\end{aligned}$$

```
n = "1071"
def OctToDec(n):
    return int(n, 8)

print("Octal " + n + " to decimal:", OctToDec(n))
#Output: Octal 1071 to decimal: 569
```

```
const n = "1071";
function OctToDec(n) {
    return parseInt(n, 8);
}
console.log("Octal " + n + " to decimal:", OctToDec(n));
//Output: Octal 1071 to decimal: 569
```

# Number Base Conversions

## ❖ Decimal to Octal

8	330		
8	41	2	↑
	5	1	

→

```
def decToOct(n):  
    return oct(n).replace("0o", "")
```

```
num = 330  
print("Decimal " + str(num) + " to octal:", decToOct(num))  
#Output: Decimal 330 to octal: 512
```

```
function decToOct(n) {  
    return (n).toString(8);  
}  
var num = 330;  
console.log("Decimal " + num + " to octal:", decToOct(num));  
//Output: Decimal 330 to octal: 512
```

# Number Base Conversions

## ❖ Hexadecimal to Decimal

Numeral	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Used For	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Hexadecimal:  $(2B4)_{16}$

Decimal:  $2 \times 16^2 + 11 \times 16^1 + 4 \times 16^0 = 512 + 176 + 4 = (692)_{10}$

```
hex_num = "2B4"
def hexToDec(n):
    return int(n, 16)

print("Hexadecimal " + hex_num + " to decimal:", hexToDec(hex_num))
#Output: Hexadecimal 2B4 to decimal: 692
```

```
const hex_num = "2B4";
function hexToDec(n) {
    return parseInt(n, 16);
}
console.log("Hexadecimal " + hex_num + " to decimal:", hexToDec(hex_num));
//Output: Hexadecimal 2B4 to decimal: 692
```



# Number Base Conversions

## ❖ Decimal to Hexadecimal

16	798			
16	49	14	→ E	
	3	1		

Diagram illustrating the conversion of decimal 798 to hexadecimal 31E. The table shows the division steps: 798 divided by 16 gives quotient 49 and remainder 14 (E); 49 divided by 16 gives quotient 3 and remainder 1. The remainders are read from bottom to top to form the hexadecimal value 31E. Arrows indicate the flow of the division and the final reading of the remainders.

```
def decToHex(n):  
    return hex(n).replace("0x", "")
```

```
num = 798
```

```
print("Decimal " + str(num) + " to hexadecimal:", decToHex(num))
```

```
#Output: Decimal 798 to hexadecimal: 31E
```

```
function decToHex(n) {  
    return n.toString(16);  
}
```

```
var num = 798;
```

```
console.log("Decimal " + num + " to hexadecimal:", decToHex(num));
```

```
//Output: Decimal 798 to hexadecimal: 31E
```

# Why different number bases?

- ❖ Translating between **decimal** (common for humans) and **binary** formats (for computers) can be difficult for large numbers.
  - $(512)_{10} = (10\ 0000\ 0000)_2 = (1000)_8 = (200)_{16}$
  - $(999)_{10} = (11\ 1110\ 0111)_2 = (1747)_8 = (3E7)_{16}$
- ❖ **Octal** or **hexadecimal** is relatively easy for humans to read and can be easily translated to binary format for computers.
  - These are **compressed representations of binary**.

# Why different number bases?

## ❖ Octal:

- Popular for **older computers (1970s)**
- **UNIX file system** permissions
- **Octal** digit represents **3 binary digits** ( $8 = 2^3$ )

## ❖ Hexadecimal:

- Useful for **decimal to binary conversion**
- Representation of **html colours (white is #FFFFFF)**
- **Hexadecimal** digit represents **4 binary digits** ( $16 = 2^4$ )

# Why different number bases?

- ❖ **Main objective:** to have a **group of bits** that should **match** the **decimal number system** which is human-understandable.
  - How many binary digits we need to group to form the number system which should satisfy human needs.
    - Single bit: represents 0 or 1
    - Two bits:  $2^2 = 4$  digits
    - Three bits:  $2^3 = 8$  digits (0 – 7, misses 8 and 9)
    - Four bits:  $2^4 = 16$  digits (0 – 9, A, B, C, D, E, F)
- ❖ **Hexadecimal minimizes** the **number of digits** we need to use, **reducing calculation time**.
- ❖ What about alphabets and special characters? Different character encoding standards eg. ASCII, Unicode, ...

# Binary to Hexadecimal

- ❖ Convert to **decimal** and then to **hexadecimal**

$$(110010111011100)_2 = (26076)_{10} = (65DC)_{16}$$

- ❖ **Direct method:** divide in **groups of four** (starting from right for integer part and start from left for fraction part)

$$(0110 \ 0101 \ 1101 \ 1100)_2$$



$$1100 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 8 + 4 = 12 = C$$

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 1 = 13 = D$$

$$0101 = 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 1 = 5$$

$$0110 = 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 4 + 2 = 6$$

$$(110010111011100)_2 = (65DC)_{16}$$

# What do you get when you convert the binary number 10011001 to decimal?

A. 150

**B. 153**

C. 154

D. 158

The only odd number in the list.

If the last digit in the binary number is 1, it will translate to  $1 \times 2^0 = 1$ , which when added to the other numbers (all of which are multiplied with powers of two and hence even) should yield an odd number.

Similarly, if the binary number ends in 0, then the decimal number is even.

# Why do we use the hexadecimal number system?

## A. Readability

- a. Hexadecimal uses digits which resemble usual base-10; easier to decide promptly how big a number like e7 is as opposed to 11100111 (231 in base-10).

## B. Higher information density

- a. With 2 hexadecimal digits, we can express any number from 0 to 255. To do the same in binary, we need 8 digits.

## C. Converts easily into bytes

- a. 1 byte = 8 bits, with  $2^8$  combinations. Hex or base-16 ( $=2^4$ ) represents exactly half a byte. Each byte can be fully represented by two hexadecimal digits.



Why do we not use base-256 ( $=2^8$ )? Then 1 digit would be exactly one byte. However, readability would be the issue.

	Base	Readable	Converts easily into bytes	Good compression
$2^2$	4	✓	✓	
$2^3$	8	✓		
$2^4$	16	✓	✓	✓
$2^5$	32	✓		✓
$2^6$	64	✓		✓
$2^7$	128			✓
$2^8$	256		✓	✓

# Binary Logic




# Binary Logic

- ❖ Binary System, simpler, less intuitive to humans, easier to implement with electronics.
- ❖ Developed by George Boole (1815–1864) – **Boolean algebra**
- ❖ Binary logic begins with statements containing variables (say X and Y) – e.g. “The switch is open” or “There is liquid water on Mars.”
- ❖ Variable representing the statement can be assigned either of **two values**, 1 or 0, depending on whether the statement is true (usually 1) or false. (usually 0).

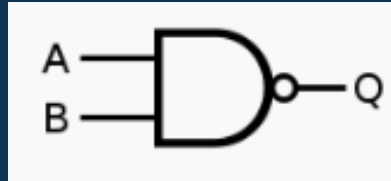


# Binary Operations



- ❖ Using binary, logic calculations can be done with simple electronic circuits based on transistors.
  - ❖ **Logic gate:** electronic circuit that can perform logical calculations.
  - ❖ A logic gate has **one or more inputs** and generates **an output**.
  - ❖ **Boolean algebra** is used to express mathematically a **logic circuit**.
  - ❖ Binary logic permits **three operations** to be performed, **AND, OR, and NOT**.
- 

# What is the truth table of this logic gate?



1.

Input		Output
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

2.

Input		Output
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

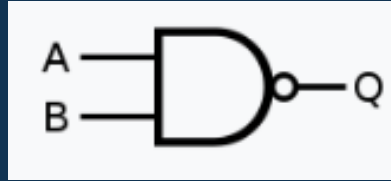
3.

Input		Output
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

4.

Input		Output
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

# What is the truth table of this logic gate?



1.

Input		Output
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

2. ✓

Input		Output
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

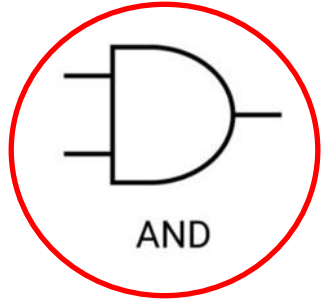
3.

Input		Output
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

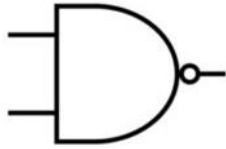
4.

Input		Output
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

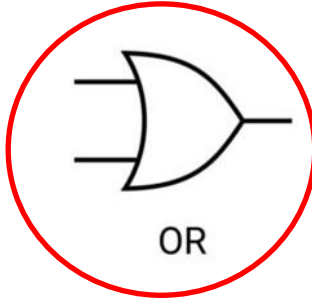
# Logic Gate Symbols



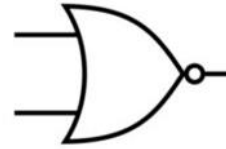
AND



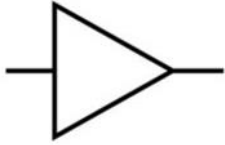
NAND



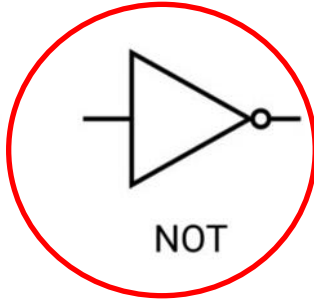
OR



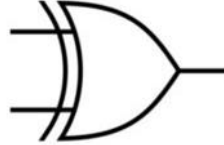
NOR



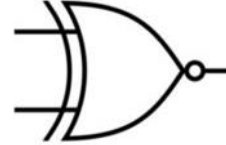
BUFFER



NOT



XOR



XNOR

## Truth Tables

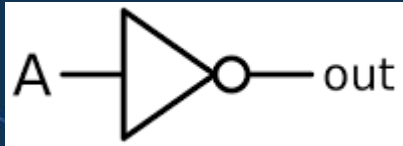
Input		Output
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1



# NOT Gate

- ❖ In electronics, also called an **inverter**.
- ❖ NOT gate implements **logical negation** (denoted by the **bubble** near output)
- ❖ **One input only**, **outputs** is the inverted input.

NOT gate



Truth table

Input	Output
A	NOT A
0	1
1	0

Boolean algebra

$\bar{A}$  or  $A'$

```
#NOT gate
def NOT(a):
    return not a
print('NOT(1):',NOT(1))
print('NOT(0):',NOT(0))
#Output: false, true
```

```
//NOT gate
function NOT(a) {
    return !a;
}
console.log('NOT(1):', NOT(1));
console.log('NOT(0):',NOT(0));
//Output: false, true
```

# AND Gate

- ❖ AND gate implements **logical conjunction** ( $\wedge$ ), or **intersection**
- ❖ **Multiple inputs**, output is high (1) if all inputs are high (1).
- ❖ AND gate finds the **minimum** between two binary digits

AND gate



Truth table

Input		Output
A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Boolean algebra

$A \cdot B$  or  $A \wedge B$

$A \& B$

```
#AND gate
def AND (a, b):

    if a == 1 and b == 1:
        return True
    else:
        return False

print('1 AND 0',AND(1, 0))
print('1 AND 1',AND(1, 1))
#Output: false, true
```

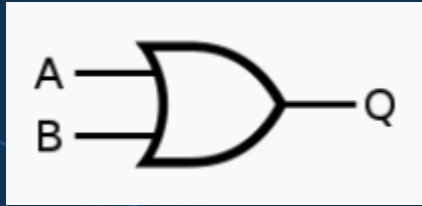
```
//AND gate
function AND(a, b) {
    if (a === 1 && b === 1) {
        return true;
    } else {
        return false;
    }
}

console.log('1 AND 0:',AND(1, 0));
console.log('1 AND 1:',AND(1, 1));
//Output: false, true
```

# OR Gate

- ❖ OR gate implements **logical disjunction** ( $\vee$ )
- ❖ **Multiple inputs**, output is high (1) if even one inputs is high (1).
- ❖ OR gate finds the **maximum** between two binary digits

OR gate



Truth table

Input		Output
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Boolean algebra

$$A + B \text{ or } A \vee B$$

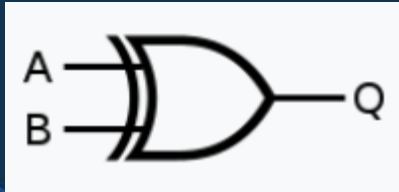
```
#OR gate
def OR(a, b):
    if a == 1 or b == 1:
        return True
    else:
        return False
print('0 AND 0',OR(0, 0))
print('0 AND 1',OR(0, 1))
#Output: false, true
```

```
//OR gate
function OR(a, b) {
    if (a === 1 || b === 1) {
        return true;
    } else {
        return false;
    }
}
console.log('0 OR 0:',OR(0, 0));
console.log('0 OR 1:',OR(0, 1));
//Output: false, true
```

# XOR Gate

- ❖ XOR (Exclusive OR) gate implements **exclusive disjunction**.
- ❖ Gives a true (1 or HIGH) output when the **number of true inputs is odd**.

XOR gate



Truth table

Input		Output
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Boolean algebra

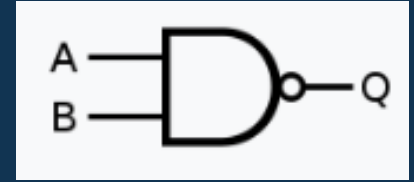
$$A \oplus B \text{ or } A \nabla B$$

```
#XOR gate
def XOR (a, b):
    if a != b:
        return True
    else:
        return False
```

```
print('1 XOR 1:',XOR(1,1))
print('1 XOR 0:',XOR(1,0))
#Output: false, true
```

```
//XOR gate
function XOR(a, b) {
    if (a !== b) {
        return true;
    } else {
        return false;
    }
}
console.log('1 XOR 1:',XOR(1, 1));
console.log('1 XOR 0:',XOR(1, 0));
//Output: false, true
```

# What is the truth table of this logic gate?



1. AND

Input		Output
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

2. NAND

Input		Output
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

3. NOR

Input		Output
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

4. XNOR


Input		Output
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

# Bitwise operations





# Bitwise operation

- ❖ Operates on a bit string, a bit array or a binary numeral at the level of its individual bits.
  - ❖ Fast and simple action, basic to the higher-level arithmetic operations and directly supported by the processor.
  - ❖ Most bitwise operations are presented as two-operand instructions where the result replaces one of the input operands.
- 





# What is the output of the expression $000111 \mid 01010$ ?

A. 000010

B. 010111

C. 101010

D. None





# What is the output of the expression $00011 \mid 01010$ ?

A. 000010

**B. 010111**

C. 101010

D. None





# Which bitwise operator is used to swap two numbers?

- A. Bitwise left shift (<<)
- B. Bitwise right shift (>>)
- C. Bitwise OR |
- D. Bitwise XOR ^





# Which bitwise operator is used to swap two numbers?

- A. Bitwise left shift (<<)
- B. Bitwise right shift (>>)
- C. Bitwise OR |
- D. Bitwise XOR ^**



What will be the output of the following Python code if  $a=10$  and  $b=20$ ?

- A. 10 20
- B. 10 10
- C. 20 10
- D. 20 20

```
a=10
b=20
a=a^b
b=a^b
a=a^b
print(a,b)
```

What will be the output of the following Python code if  $a=10$  and  $b=20$ ?

- A. 10 20
- B. 10 10
- C. 20 10**
- D. 20 20

```
a=10  
b=20  
a=a^b  
b=a^b  
a=a^b  
print(a,b)
```

# Bitwise operators

Operator	Example	Meaning	Description
&	a & b	Bitwise AND	Sets each bit to 1 if both bits are 1
	a   b	Bitwise OR	Sets each bit to 1 if one of two bits is 1
^	a ^ b	Bitwise XOR (exclusive OR)	Sets each bit to 1 if only one of two bits is 1
~	~a	Bitwise NOT	Inverts all the bits
<<	a << n	Bitwise left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	a >> n	Bitwise right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off



# Bitwise AND

- ❖ Bitwise AND (&) performs **logical conjunction** on corresponding bits.
- ❖ Result is an **intersection** of the operator's arguments. Returns a one only when **both bits are switched on**.

a = 10 011 100 (decimal 156)  
b = 110 100 (decimal 52)

---

a & b = 00 010 100 (decimal 20)

a = 10 10 (decimal 10)  
b = 01 00 (decimal 4)

---

a & b = 00 00 (decimal 0)

1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---

AND (&)

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

$$(a \& b)_i = a_i \times b_i$$

# Bitwise OR

- ❖ Bitwise OR (|) performs **logical disjunction** on corresponding bits of its operands.
- ❖ Result is a **union** of the operator's arguments. Returns a one if **at least one of them is switched on**.

a = 10 011 100 (decimal 156)  
b = 110 100 (decimal 52)

---

a | b = 10 111 100 (decimal 188)

a = 10 10 (decimal 10)  
b = 01 00 (decimal 4)

---

a | b = 11 10 (decimal 14)

1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---

OR (|)

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

$$(a \mid b)_i = a_i + b_i - (a_i \times b_i)$$

# Bitwise XOR

- ❖ Bitwise XOR ( $\wedge$ ) performs **exclusive disjunction** on bit pairs.
- ❖ Returns 1 if **one** of the **bits is 1** and the **other is 0** else returns false.

a = 10 011 100 (decimal 156)  
b = 110 100 (decimal 52)

---

a  $\wedge$  b = 10 101 000 (decimal 168)

a = 10 10 (decimal 10)  
b = 01 00 (decimal 4)

---

a  $\wedge$  b = 11 10 (decimal 14)

1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---

XOR ( $\wedge$ )

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---



# Bitwise NOT

- ❖ Bitwise NOT, or bitwise complement, is a unary operation (i.e. only one input)
- ❖ Performs logical negation on each bit, 0 becomes 1, and 1 becomes 0
- ❖ Bitwise negation inverts sign bit (negative -x written using bit pattern for (x-1))
- ❖ Bitwise NOT is the same as  $-x - 1$

NOT 1010 (decimal 10)  
= - 1011 (decimal -11)

NOT 10101010 (decimal 170)  
= -10101011 (decimal -171)

# Bitwise Left Shift

- ❖ Bitwise shift operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two.
- ❖ Bitwise left shift: shifts bits of the number to left and fills 0 on voids right.
- ❖ Similar effect as of multiplying the number with some power of two.

$a = 101$  (decimal 5)

$a = 100111$  (decimal 39)

$a \ll 1 = 1010$  (decimal 10)

$a \ll 1 = 1001110$  (decimal 78)

$a \ll 2 = 10100$  (decimal 20)

$a \ll 2 = 10011100$  (decimal 156)

$a \ll 3 = 101000$  (decimal 40)

$a \ll 3 = 100111000$  (decimal 312)



# Bitwise Right Shift

- ❖ **Bitwise right shift**: shifts bits of the number to right and fills 0 on voids left (fills 1 in the case of a negative number)
- ❖ Similar effect as of **dividing** the number **with some power of two**.

$a = 1010$  (decimal 10)

$a \gg 1 = 0101$  (decimal 5)

$a \gg 2 = 010$  (decimal 2)

$a \gg 3 = 001$  (decimal 1)

$a = 10011101$  (decimal 157)

$a \gg 1 = 1001110$  (decimal 78)

$a \gg 2 = 100111$  (decimal 39)

$a \gg 3 = 10011$  (decimal 19)



# Bitwise operations in Python

```
# Python program to show bitwise operators
```

```
a = 10
```

```
b = 4
```

```
print("a = ", a, '(decimal)', bin(a), '(binary)')
```

```
print("b = ", b, '(decimal)', bin(b), '(binary)')
```

```
# Print bitwise AND operation
```

```
print("AND: a & b =", a & b, '(decimal)', bin(a & b), '(binary)')
```

```
# Print bitwise OR operation
```

```
print("OR: a | b =", a | b, '(decimal)', bin(a | b), '(binary)')
```

```
# Print bitwise NOT operation
```

```
print("NOT: ~a =", ~a, '(decimal)', bin(~a), '(binary)')
```

```
# Print bitwise XOR operation
```

```
print("XOR: a ^ b =", a ^ b, '(decimal)', bin(a ^ b), '(binary)')
```

```
# Print bitwise left shift operation
```

```
print("Left shift: a << 1 =", a << 1, '(decimal)', bin(a << 1), '(binary)')
```

```
# Print bitwise right shift operation
```

```
print("Left shift: a >> 1 =", a >> 1, '(decimal)', bin(a >> 1), '(binary)')
```

```
a = 10 (decimal) 0b1010 (binary)
```

```
b = 4 (decimal) 0b100 (binary)
```

```
AND: a & b = 0 (decimal) 0b0 (binary)
```

```
OR: a | b = 14 (decimal) 0b1110 (binary)
```

```
NOT: ~a = -11 (decimal) -0b1011 (binary)
```

```
XOR: a ^ b = 14 (decimal) 0b1110 (binary)
```

```
Left shift: a << 1 = 20 (decimal) 0b10100 (binary)
```

```
Left shift: a >> 1 = 5 (decimal) 0b101 (binary)
```



# Bitwise operations in JavaScript

```
//Bitwise operators
```

```
let a = 10;  
let b = 4;
```

```
console.log("a = ", a, '(decimal)', a.toString(2), '(binary)');  
console.log("b = ", b, '(decimal)', b.toString(2), '(binary)');
```

```
// Print bitwise AND operation
```

```
console.log("AND: a & b =", a & b, '(decimal)', (a & b).toString(2), '(binary)');
```

```
//Print bitwise OR operation
```

```
console.log("OR: a | b =", a | b, '(decimal)', (a | b).toString(2), '(binary)');
```

```
//Print bitwise NOT operation
```

```
console.log("NOT: ~a =", ~a, '(decimal)', (~a).toString(2), '(binary)');
```

```
// Print bitwise XOR operation
```

```
console.log("XOR: a ^ b =", a ^ b, '(decimal)', (a ^ b).toString(2), '(binary)');
```

```
// Print bitwise left shift operation
```

```
console.log("Left shift: a << 1 =", a << 1, '(decimal)', (a << 1).toString(2), '(binary)');
```

```
// Print bitwise right shift operation
```

```
console.log("Left shift: a >> 1 =", a >> 1, '(decimal)', (a >> 1).toString(2), '(binary)');
```

a = 10 (decimal) 1010 (binary)

b = 4 (decimal) 100 (binary)

AND: a & b = 0 (decimal) 0 (binary)

OR: a | b = 14 (decimal) 1110 (binary)

NOT: ~a = -11 (decimal) -1011 (binary)

XOR: a ^ b = 14 (decimal) 1110 (binary)

Left shift: a << 1 = 20 (decimal) 10100 (binary)

Left shift: a >> 1 = 5 (decimal) 101 (binary)



# What is the output of the expression $000111 \mid 01010$ ?

A. 000010

B. **010111**      Bitwise OR operation

C. 101010

D. None



# Which bitwise operator is used to swap two numbers?

- A. Bitwise left shift (<<)
- B. Bitwise right shift (>>)
- C. Bitwise OR |
- D. **Bitwise XOR ^**

The XOR of two numbers a and b returns a number that has all the bits as 1 wherever bits of a and b differ.

$1010 \wedge 0101 = 1111$   
 $0111 \wedge 0101 = 0010$

# What will be the output of the following Python code if $a=10$ and $b=20$ ?

```
a=10
b=20
a=a^b
b=a^b
a=a^b
print(a,b)
```

A. 10 20

B. 10 10

C. **20 10**

D. 20 20

XOR is used to swap two numbers

In binary

$a = 01010$

$b = 10100$

$a = a \wedge b = 11110$

$b = a \wedge b = 01010$

$a = a \wedge b = 10100$

In decimal,  $a = 20$ ,  $b = 10$

# Bit manipulation



# Bit manipulation

Programming languages work with objects or variables, rather than bits. However, direct bit manipulation improves performance and reduces error.

- ❖ Low-level device control
- ❖ Error detection and correction algorithms
- ❖ Data compression
- ❖ Encryption algorithms
- ❖ Optimization

Bit manipulation is also a common topic in coding interviews. The closer your role is to machine level, the more bit manipulation questions you'll encounter.


# Bit operators in tasks

- ❖ **Bit Manipulation:** Setting, clearing, or toggling specific bits within an integer
- ❖ **Masking:** Extracting specific bits or groups of bits from an integer using bitwise AND. Bitmasks are used to apply filtering or toggle multiple options simultaneously.
- ❖ **Bit-Level Flags:** Efficiently representing and manipulating sets of Boolean flags, used to represent multiple Boolean values compactly using individual bits.
- ❖ **Bitwise Operations in Encryption:** Implementing cryptographic algorithms
- ❖ **Low-Level Hardware Control:** Interfacing with hardware registers in embedded systems



# Data compression techniques






Using run-length encoding, how  
would the string  
'EEEEENNCCRRRRYPPTTT' be  
encoded as?

- A. 4E3N2C3RY2P3P
- B. 4E1N2C3RY2P3Y
- C. 4E2N2C3R1Y2P3T
- D. 3E2N2C3RY2P3T





Using run-length encoding, how  
would the string  
'EEEENNCCRRRRYPPTTT' be  
encoded as?

A. 4E3N2C3RY2P3P

B. 4E1N2C3RY2P3Y


**C. 4E2N2C3R1Y2P3T**

D. 3E2N2C3RY2P3T





# Which of the following statements about data compression is true?

- A. Data compression always requires converting data to a different format.
  - B. Lossy compression ensures the integrity of the original data.
  - C. Lossless compression sacrifices some data to achieve greater compression.
  - D. Many modern compression software systems use a combination of lossless and lossy methods.
- 

# Which of the following statements about data compression is true?

- A. Data compression always requires converting data to a different format.
- B. Lossy compression ensures the integrity of the original data.
- C. Lossless compression sacrifices some data to achieve greater compression.
- D. Many modern compression software systems use a combination of lossless and lossy methods.**

# Data Compression

**Data compression** is the process by which size of shared or stored data is reduced (reduces the number of bits), e.g. hexadecimal to binary.

Reduction depends on


- ❖ Amount of redundancy in the data or repeated information that can be removed from original data
- ❖ Method used to compress
- ❖ **Run-length encoding (RLE)** – replaces repeating data (colours in an image, letters in a document with a run with the number and value of repeated data, e.g. string FFFFFIIGGG can be replaced by 5F2I3G)
- ❖ **LZW (lossless) compression** – Replaces data with symbols, frequently occurring patterns with shorter codes, compress text and images (notably GIFs, TIFFs)



# Lossy & Lossless Compression

Lossy Compression	Lossless Compression
Eliminate the data which is not noticeable, discard some information	Does not eliminate the data which is not noticeable, all information is preserved
A file does not restore or rebuilt in its original form.	A file can be restored in its original form.
Data's quality is compromised.	Does not compromise the data's quality.
Reduces the size of data	Does not reduce the size of data.
Used in images, audio, video.	Used in databases, text, images, sound.
Has more data-holding capacity.	Has less data-holding capacity.
Irreversible compression.	Reversible compression.





Using run-length encoding, how  
would the string  
'EEEENNCCRRRRYPPTTT' be  
encoded as?

A. 4E3N2C3RY2P3P

B. 4E1N2C3RY2P3Y

**C. 4E2N2C3R1Y2P3T**

D. 3E2N2C3RY2P3T



# Which of the following statements about data compression is true?

- A. Data compression always requires converting data to a different format.
- B. Lossy compression ensures the integrity of the original data.
- C. Lossless compression sacrifices some data to achieve greater compression.
- D. Many modern compression software systems use a combination of lossless and lossy methods.**

# Which of the following statements about data compression is true?

- ❖ Many modern compression software systems utilize a combination of lossless and lossy methods. This approach allows for achieving the benefits of both types of compression techniques, depending on the specific characteristics of the data being compressed.
- ❖ Data compression does not always involve converting data to a different format, and compression can be achieved while preserving the original format.
- ❖ Lossless compression methods aim to preserve all the original data, whereas lossy compression methods sacrifice some data to achieve greater compression.
- ❖ Lossy compression methods inherently involve sacrificing some data and, therefore, do not ensure the integrity of the original data.

# Portfolio Assignment





# Portfolio Assignment

- ❖ Feel free to adjust according to your interests and specific track (data science, web development, or software engineering)

**Submit your solutions here!**

# Portfolio Assignment: Data Science

Choose one or more of the following topics to focus on:

- Find elements in an array using XOR.
- ❖ Find repeating elements in an array of size  $n$
- ❖ Find all odd occurring elements in an array
- Check if binary representation of a number is palindrome or not.
- Swapping variables without using a third variable (shown in this lecture)

# Portfolio Assignment: Web development

- Color channel values are simply integers that range from 0 (0x00) to 255 (0xFF). Using bitwise operators (a) check whether a palette has the blue pixel or not, (b) insert a new value (0xEE) into the green channel of an existing color palette.






# Portfolio Assignment: Software development

- A small car has 3 doors: 2 front doors (X, Y) and a hatchback (Z). The internal light of the car (L) is on when any of the car doors is open. Create and test a logic gates circuit to control the internal light (L) of the car based on whether a door (X, Y, or Z) is open (1) or closed (0).





# Portfolio Assignment: Software development

- ❖ A bitmask is nothing more than a number that defines which bits are on and off, or a binary string representation of the number.
  - ❖ Bitmask dynamic programming uses bitmasks, to keep track of our current state in a problem.
  - ❖ Travelling salesman problem: Given a list of  $N$  cities and the distances between each pair of cities, what is the shortest possible route that the salesperson visits each city exactly once and returns to the city that they started from?
  - ❖ Can be done using permutations, which slows down for large  $N$ .
  - ❖ Use bitmasks and a recursive function instead.
- 



# Portfolio Assignment

- ❖ Use HTML, CSS, and JavaScript to build the frontend of any app, focusing on creating a clean and intuitive user interface
- ❖ Implement the necessary mathematical calculations and algorithms using JavaScript or a backend language of your choice (e.g., Python, Java)
- ❖ Provide clear instructions on how to use the app and explain the concepts being demonstrated
- ❖ Host the project on GitHub Pages or a similar platform and include a brief description of the app in the README file



**Submit your solutions here!**



# Portfolio Assignment

## ❖ Evaluation Criteria:

- Correctness and accuracy of the calculations and visualisations
- User interface design and ease of use
- Code quality, organisation, and documentation
- Clarity and effectiveness of the project description and instructions
- Creativity and originality in applying concepts to practical problems

# References

- ❖ Bitwise operations - <https://realpython.com/python-bitwise-operators>
- ❖ <https://wiki.python.org/moin/BitwiseOperators>
- ❖ <https://www.geeksforgeeks.org/python-bitwise-operators/>
- ❖ [Bit manipulation](#)
- ❖ <https://cp-algorithms.com/algebra/bit-manipulation.html>
- ❖ [Data compression](#)



# CoGrammar

## Q & A SECTION

Please use this time to ask any questions relating to the topic, should you have any.



# Thank you for attending



Department  
for Education

CoGrammar

