Here are the binary representations of the given numbers:

1. **512** = `1000000000`
2. **44** = `101100`
3. **129** = `10000001`
4. **717** = `1011001101`
5. **999** = `1111100111`

To write the name **"Mustafa"** in binary, we convert each character to its **ASCII** code, then to **8-bit binary**.

Here's the breakdown:

Letter	ASCII	Binary
M	77	01001101
u	117	01110101
s	115	01110011
t	116	01110100
a	97	01100001
f	102	01100110
a	97	01100001

Result in binary:

```
01001101 01110101 01110011 01110100 01100001 01100110 01100001
```

Here are the hexadecimal to decimal conversions:

1. **12A4** (hex) =

   1×163+2×162+10×161+4×1601 \times 16^3 + 2 \times 16^2 + 10 \times 16^1 + 4 \times 16^01×163+2×162+10×161+4×160

=4096+512+160+4=4772= 4096 + 512 + 160 + 4 = \boxed{4772}=4096+512+160+4=4772

2. **1C35** (hex) =

   1×163+12×162+3×161+5×1601 \times 16^3 + 12 \times 16^2 + 3 \times 16^1 + 5 \times 16^01×163+12×162+3×161+5×160

   =4096+3072+48+5=7221= 4096 + 3072 + 48 + 5 = \boxed{7221}=4096+3072+48+5=7221

3. **100** (hex) =

   1×162+0×161+0×1601 \times 16^2 + 0 \times 16^1 + 0 \times 16^01×162+0×161+0×160

   =256+0+0=256= 256 + 0 + 0 = \boxed{256}=256+0+0=256

4. **115C** (hex) =

   1×163+1×162+5×161+12×1601 \times 16^3 + 1 \times 16^2 + 5 \times 16^1 + 12 \times 16^01×163+1×162+5×161+12×160

   =4096+256+80+12=4444= 4096 + 256 + 80 + 12 = \boxed{4444}=4096+256+80+12=4444

Here are the decimal to hexadecimal conversions:

1. **4722** = **0x1272**
2. **7221** = **0x1C35**
3. **256** = **0x100**
4. **4444** = **0x115C**

Here are the hexadecimal to binary conversions:

---

1. 13B4 (hex):

Break each digit into 4-bit binary:

- 1 → `0001`

- 3 → `0011`
- B → `1011` (B = 11 in decimal)
- 4 → `0100`

Result: `0001 0011 1011 0100`

Binary: `0001001110110100`

---

2. A5 (hex):

- A → `1010` (A = 10 in decimal)
- 5 → `0101`

Result: `1010 0101`

Binary: `10100101`

1. What is a bit?

A **bit** (short for **binary digit**) is the **smallest unit of data** in computing. It can only have one of two possible values:

- **0** (off, false, low)
- **1** (on, true, high)

Computers use bits to represent all types of data and perform operations using combinations of bits.

🔷 **Example:**

The number **5** in binary is `101`, which consists of 3 bits.

---

2. What is a crumb?

A **crumb** is an informal term that refers to **2 bits**. While not commonly used in standard computing language, it appears in educational or theoretical contexts.

🔷 **Example:**

The binary number `11` is a single crumb, representing the value **3** in decimal.

---

3. What is a nibble?

A **nibble** is a group of **4 bits**. It's half of a byte and often used to represent **a single hexadecimal digit** (0–15).

🔷 **Example:**

The binary `1010` is a nibble and corresponds to the hexadecimal digit **A**.

---

4. What is a byte?

A **byte** is a group of **8 bits** and is a fundamental unit of data in computing. Most modern systems treat a byte as the basic addressable memory unit.

🔷 **Example:**

`01001101` is one byte and represents the ASCII character **'M'**.

---

5. How many bits in a byte?

There are **8 bits** in **1 byte**.

---

6. How many crumbs in a byte?

Since a crumb is **2 bits**, and a byte is **8 bits**:

8 bits2 bits per crumb=4 crumbs\frac{8 \text{ bits}}{2 \text{ bits per crumb}} = 4 \text{ crumbs}

2 bits per crumb8 bits=4 crumbs

So, there are **4 crumbs** in a byte.

---

7. How many nibbles in a byte?

Since a nibble is **4 bits**:

8 bits4 bits per nibble=2 nibbles\frac{8 \text{ bits}}{4 \text{ bits per nibble}} = 2 \text{ nibbles}

4 bits per nibble8 bits=2 nibbles

So, there are **2 nibbles** in a byte.

---

8. How many crumbs in a nibble?

A nibble is **4 bits**, and a crumb is **2 bits**:

42=2\frac{4}{2} = 2

24=2

So, a nibble contains **2 crumbs**.

---

9. What is the lower nibble?

In an 8-bit byte, the **lower nibble** refers to the **least significant 4 bits** (the rightmost 4 bits).

🔷 **Example:**

Byte = `10101101`

Lower nibble = `1101` (right 4 bits)

This is important in operations involving bit masking or when handling hexadecimal digits.

---

10. What is the upper nibble?

The **upper nibble** refers to the **most significant 4 bits** (the leftmost 4 bits) of a byte.

🔷 **Example:**

Byte = `10101101`

Upper nibble = `1010` (left 4 bits)

It often determines the range or category of values.

---

11. What is LSB? And why is it called LSB? What is its value?

LSB stands for **Least Significant Bit**. It is the **rightmost bit** in a binary number and contributes the **smallest value** in the overall number. It's called LSB because a change in this bit has the **smallest impact** on the total value.

🔷 **Example:**

In binary `1101`, the LSB is the rightmost `1`, which represents 20=12^0 = 120=1

So its value is **1** in this case.

---

12. What is MSB? And why is it called MSB? What is its value?

MSB stands for **Most Significant Bit**. It is the **leftmost bit** in a binary number and contributes the **largest value**. It's called MSB because it determines the **overall scale** of the number.

🔷 **Example:**

In binary `1101`, the MSB is the leftmost `1`, which represents 23=82^3 = 823=8

So its value is **8** in this case.

In **signed binary representations**, the MSB often determines the **sign** of the number (0 = positive, 1 = negative).

To **directly convert** hexadecimal numbers to binary, we convert **each hex digit** into its **4-bit binary equivalent**.

✅ Conversion Table (Hex → Binary):

Hex	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100

Hex	Binary
D	1101
E	1110
F	1111

🔢 Direct Conversions:

1. **12A4**

   → 1 = `0001`

   → 2 = `0010`

   → A = `1010`

   → 4 = `0100`

   ✅ **Binary:** `0001 0010 1010 0100`

   ✅ **Result:** `0001001010100100`

---

1. **1C35**

   → 1 = `0001`

   → C = `1100`

   → 3 = `0011`

   → 5 = `0101`

   ✅ **Binary:** `0001 1100 0011 0101`

   ✅ **Result:** `0001110000110101`

---

1. **100**

   → 1 = `0001`

   → 0 = `0000`

   → 0 = `0000`

   ✅ **Binary:** `0001 0000 0000`

   ✅ **Result:** `000100000000`

---

1. **115C**

   → 1 = `0001`

   → 1 = `0001`

   → 5 = `0101`

   → C = `1100`

   ✅ **Binary:** `0001 0001 0101 1100`

   ✅ **Result:** `0001000101011100`

Here are the **direct conversions** of the given binary numbers to **hexadecimal**. We'll group the binary into 4-bit chunks (nibbles), then convert each nibble to its hexadecimal equivalent.

✅ Conversion Table (Binary → Hex):

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D

Binary	Hex
1110	E
1111	F

🔢 Direct Conversions:

1. **0001 0010 1010 0100**

   → 0001 = **1**

   → 0010 = **2**

   → 1010 = **A**

   → 0100 = **4**

   ✅ **Hex: 12A4**

---

1. **0001 1100 0011 0101**

   → 0001 = **1**

   → 1100 = **C**

   → 0011 = **3**

   → 0101 = **5**

   ✅ **Hex: 1C35**

---

1. **0001 0000 0000**

   → Group: `0001 0000 0000`

   → Add leading 0 to make full nibbles: `0001 0000 0000` → no change needed

   → 0001 = **1**

   → 0000 = **0**

   → 0000 = **0**

   ✅ **Hex: 100**

---

1. **0001 0001 0101 1100**

   → 0001 = **1**

→ 0001 = **1**

→ 0101 = **5**

→ 1100 = **C**

✅ **Hex: 115C**

To convert **octal numbers** to **decimal**, you expand the number using **base-8 positional values**:

> Octal number: digits are multiplied by powers of 8 from right to left.

✅ 1. Convert 100 (Octal) to Decimal:

1×82+0×81+0×80=64+0+0=641 \times 8^2 + 0 \times 8^1 + 0 \times 8^0 = 64 + 0 + 0 = \boxed{64}

1×82+0×81+0×80=64+0+0=64

✅ 2. Convert 512 (Octal) to Decimal:

5×82+1×81+2×80=320+8+2=3305 \times 8^2 + 1 \times 8^1 + 2 \times 8^0 = 320 + 8 + 2 = \boxed{330}

5×82+1×81+2×80=320+8+2=330

To convert **decimal to octal**, we repeatedly **divide the decimal number by 8** and record the **remainders**, then **read them from bottom to top**.

✅ 1. Convert 64 (Decimal) to Octal:

Divide by 8:

```
64 ÷ 8 = 8    remainder 0
8 ÷ 8  = 1    remainder 0
```

```
1 ÷ 8  = 0    remainder 1
```

🔶 **Read from bottom up: 1 0 0**

✅ **Octal: 100**

✅ 2. Convert 330 (Decimal) to Octal:

Divide by 8:

```
330 ÷ 8 = 41   remainder 2
41 ÷ 8  = 5    remainder 1
5 ÷ 8   = 0    remainder 5
```

🔶 **Read from bottom up: 5 1 2**

✅ **Octal: 512**

To **directly convert octal to binary**, convert **each octal digit** to its **3-bit binary equivalent**, since:

> 🧠 1 octal digit = 3 binary bits

✅ Octal to Binary Table:

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101

Octal	Binary
6	110
7	111

🔢 1. Convert 100 (Octal) to Binary:

Digits: **1 – 0 – 0**

→ 1 = `001`

→ 0 = `000`

→ 0 = `000`

✅ **Binary:** `001 000 000` → `001000000`

🔢 2. Convert 512 (Octal) to Binary:

Digits: **5 – 1 – 2**

→ 5 = `101`

→ 1 = `001`

→ 2 = `010`

✅ **Binary:** `101 001 010` → `101001010`

To **directly convert binary to octal**, follow these steps:

> 🧠 Group the binary digits in 3s from right to left, then convert each 3-bit group to its octal value.

If the number of bits isn't a multiple of 3, **add leading zeros** to the left to complete the first group.

🔢 1. Convert 0100 0000 (Binary) to Octal:

Remove the space and group into 3s from right:

- Binary: `01000000`

- Add leading 0 to make groups of 3: `000 100 000 0` → need to make it: `000 100 000`
- Final groups: `000` `100` `000`

Now convert each group:

- `000` → 0
- `100` → 4
- `000` → 0

✅ **Octal: 040** → we usually write it as **40**

🔢 2. Convert 0001 0100 1010 (Binary) to Octal:

Remove spaces: `000101001010`

Group from right into 3s:

- `000` `101` `001` `010`

Convert:

- `000` → 0
- `101` → 5
- `001` → 1
- `010` → 2

✅ **Octal: 0512**

✅ Final Answers:

- `0100 0000` → **40** (octal)
- `0001 0100 1010` → **512** (octal)

✅ 1. What is a network?

A **network** is a group of **connected computers or devices** that can **communicate and share resources** (such as files, printers, or internet access). The connection can be **wired** (using cables) or **wireless** (using radio signals).

🔷 **Example:**

A company's office with multiple computers connected to a central server to share data and printers forms a **network**.

✅ 2. Why do we need a network?

We need networks to:

- **Share resources** (printers, files, applications)
- **Communicate efficiently** (email, messaging, video conferencing)
- **Access centralized data** or servers
- **Improve productivity** through collaboration
- **Access the internet**

🔷 **Example:**

In a school, all student computers are connected to a network so they can access the internet, submit homework online, and use shared educational software.

✅ 3. What is LAN vs WAN?

- **LAN (Local Area Network):**
  - Covers a **small geographic area** (like a home, office, or school)
  - High speed, low latency
  - Owned and maintained **privately**

🔷 *Example:* A Wi-Fi network in your house

- **WAN (Wide Area Network):**
  - Covers a **large geographic area** (like cities, countries)
  - Slower than LAN and may use public infrastructure
  - Often managed by **internet service providers**

🔷 *Example:* The **Internet** is the biggest example of a WAN

✅ 4. How do computers communicate together?

Computers communicate using a combination of:

- **Hardware**: Network Interface Cards (NICs), routers, switches, cables, antennas
- **Protocols**: Rules and standards that define how data is formatted and transmitted (e.g., TCP/IP)
- **IP addresses**: Unique identifiers assigned to devices

Data is broken into **packets**, sent over the network, and **reassembled** at the receiving device.

🔷 **Example:**

When you open a website, your computer sends a request packet to a server using the **HTTP protocol**, and the server responds with the webpage data.

✅ 5. What is Ethernet?

Ethernet is a **wired networking technology** used in LANs. It defines how devices on the same network communicate using **cables and switches**.

It provides:

- Reliable, high-speed data transfer
- Typically uses **RJ45 cables**

🔷 **Example:**

A desktop computer in an office connected to a switch using a cable is using **Ethernet** to access the internet.

✅ 6. What is Wireless?

Wireless refers to communication between devices without physical cables. It uses **radio waves** or **infrared signals**.

Wireless networks allow mobility and easier installation but can be less secure and slightly slower than wired connections.

🔷 **Example:**

Connecting your smartphone to the internet via **Wi-Fi** at a café.

✅ 7. What is a Protocol?

A **protocol** is a set of **rules and standards** that determine how data is **transmitted and received** over a network. It ensures that devices can understand each other even if they are different types or from different manufacturers.

🔷 **Examples of protocols:**

- **TCP/IP** – foundational protocol of the internet
- **HTTP/HTTPS** – used for accessing web pages
- **FTP** – used for transferring files

🔷 **Example:**

When sending an email, your device uses **SMTP** (Simple Mail Transfer Protocol) to send the message to the mail server.

✅ 8. What is Wi-Fi?

Wi-Fi is a wireless technology that allows devices to connect to a **local area network (LAN)** without physical cables using **radio frequency signals**.

Wi-Fi is commonly used in homes, offices, cafes, airports, etc., and requires a **wireless router or access point**.

🔷 **Example:**

Your laptop connects to your home router via Wi-Fi to access the internet.

✅ 9. Wi-Fi stands for what?

Wi-Fi stands for **"Wireless Fidelity"**, although it originally didn't mean anything specific — it was a branding name created as a play on "Hi-Fi" (High Fidelity).

Today, it refers to **IEEE 802.11** standards that define wireless local area networking.

✅ 1. What is a Programming Language?

A **programming language** is a formal language used by **humans to write instructions** that a computer can understand and execute. It defines a set of **syntax rules** and **keywords** used to build **software, websites, mobile apps**, and more.

🔷 **Example:**

Languages like **Python**, **C++**, and **Java** are all programming languages, each with its own syntax and use cases.

✅ 2. What is Code?

Code refers to the **set of instructions written** in a programming language. It tells the computer **what to do**, such as performing calculations, displaying data, or making decisions.

🔷 **Example:**

```
print("Hello, World!")
```

This line of **code** tells the computer to print a message to the screen.

✅ 3. What is Source Code?

Source code is the **original, human-readable form of code** written by a programmer in a high-level language. It needs to be **translated (compiled or interpreted)** into machine code before a computer can execute it.

🔷 **Example:**

```
int main() {
    printf("Hello, World!");
    return 0;
```

```
}
```

This is **source code** in C that needs to be compiled before it can run.

✅ 4. What is Object Code?

Object code is the **machine-readable** version of the source code, usually the result of **compilation**. It consists of **binary instructions** that a computer's CPU can execute directly.

🔷 **Example:**

After compiling the C source code above, you get an object file like `main.o` that contains low-level binary data.

✅ 5. Why do we need Compilers or Interpreters?

We need **compilers** and **interpreters** to **translate human-written source code** into machine code, which computers can actually run.

- A **compiler** translates the **entire code at once** and creates an executable file.
- An **interpreter** translates and runs the code **line by line**, without producing a separate file.

🔷 **Example:**

- C uses a **compiler**
- Python uses an **interpreter**

✅ 6. When is a language considered Fast?

A programming language is considered **fast** when:

- Its compiled code runs **efficiently** on the CPU
- It provides **low-level control** over memory and hardware
- It has **less runtime overhead**

🔷 **Example:**

C and C++ are fast languages used in performance-critical systems like operating systems and game engines.

✅ 7. When is a language considered Slow?

A language is considered **slow** when:

- It has **high-level abstractions** that trade performance for simplicity
- It uses **interpreters** instead of compilers
- It depends on a **virtual machine or runtime environment**

🔷 **Example:**

Python is considered slower than C because it runs via an interpreter and abstracts many low-level operations.

✅ 8. What is a High-Level Language?

A **high-level language** is close to **human language**, easy to read and write, and abstracts away hardware details like memory management and registers.

🔷 **Examples:** Python, Java, C#, JavaScript

🔷 **Example Code (Python):**

```
if x > 10:
    print("Large number")
```

This is readable, short, and clear.

✅ 9. What is a Low-Level Language?

A **low-level language** is close to the **machine's native language**, giving the programmer more control over memory and performance. It's harder to read and write but very efficient.

🔷 **Examples:**

- **Assembly language** (human-readable but hardware-specific)
- **Machine language** (binary code)

🔷 **Example (Assembly):**

```
MOV AX, 0001
ADD AX, 0002
```

This is much more technical and closer to the hardware.

✅ 10. When do we consider a language to be human-readable?

A language is **human-readable** when its **syntax resembles natural language** and its structure is **intuitive** and **easy to understand** by humans (especially beginners).

🔷 **Examples:** Python, JavaScript

🔷 **Example:**

```
for item in shopping_list:
    print(item)
```

This code clearly communicates its purpose to a human, even with basic knowledge.

✅ 1. What is a compiler? And how does it work?

A **compiler** is a program that **translates the entire source code** (written in a high-level language like C or C++) **into machine code** (binary code) **before execution**.

✅ How it works:

1. **Lexical Analysis** – Breaks code into tokens
2. **Syntax Analysis** – Checks grammar using parsing
3. **Semantic Analysis** – Validates meaning (e.g. type checking)
4. **Code Generation** – Translates to machine code
5. **Optimization** – Improves performance of the generated code
6. **Output** – Produces an executable file (e.g., `.exe` on Windows)

🔷 **Example:**

You write a C program `hello.c`, compile it using GCC:

```
gcc hello.c -o hello.exe
```

The compiler produces `hello.exe`, which can be run directly.

✅ 2. What is an interpreter? And how does it work?

An **interpreter** is a program that **translates and executes code line-by-line** at runtime **without producing a separate executable file**.

✅ How it works:

1. Reads a line of source code
2. Translates it to machine code or bytecode
3. Executes it immediately
4. Repeats for the next line

🔷 **Example:**

In **Python**:

```
print("Hello")
```

This line is interpreted and executed on the spot when you run:

```
python hello.py
```

✅ 3. What is an assembler?

An **assembler** is a program that **converts assembly language** (human-readable low-level code) **into machine code** (binary).

🔷 **Example:**

Assembly code:

```
MOV AX, 0001
ADD AX, 0002
```

Assembler turns this into binary instructions the CPU can execute.

✅ 4. What is a linker?

A **linker** is a tool that **combines multiple object files** (e.g., compiled functions and libraries) into a **single executable program**. It also resolves references between files.

🔷 **Example:**

If your program uses a math library, the linker adds the machine code for those math functions into your final `.exe`.

✅ 5. What is an `.exe` file?

An `.exe` **file** is a **Windows executable file** — it contains **machine code** that is ready to run on a computer. It is the final output of a compiled program.

🔷 **Example:**

When you compile `main.c`, you get `main.exe`. Double-clicking this file on Windows will execute your program.

✅ 6. What is a loader?

A **loader** is a part of the **operating system** that **loads the** `.exe` **file (or program)** from disk into **RAM**, sets up the memory, and **starts execution**.

🔷 **Example:**

When you double-click `hello.exe`, the loader loads it into memory and runs it.

✅ 7. Which is faster: Compiled or Interpreted languages?

Compiled languages are generally **faster** because:

- The code is translated once into machine code and runs directly
- There's no runtime translation overhead

🔷 **Examples:**

- **Compiled:** C, C++, Go → ✅ Fast
- **Interpreted:** Python, JavaScript → ❌ Slower

✅ 8. Does an interpreter produce an `.exe` file?

❌ **No.**

An **interpreter does not produce an** `.exe` **file**. It runs the source code directly **each time** the program is executed.

🔷 Example: Running `python app.py` every time needs Python installed.

✅ 9. Does an interpreter save machine code?

❌ **No.**

An interpreter does **not save machine code to a file**. It may generate **temporary internal code in memory**, but it is **not stored** for reuse.

🔷 Example: Each time you run a Python file, it's reinterpreted.

✅ 10. Does a compiler produce an `.exe` file?

✅ **Yes.**

A compiler produces an **executable file**, usually with an `.exe` extension on Windows or no extension on Linux/Mac.

🔷 Example:

```
gcc main.c -o main.exe
```

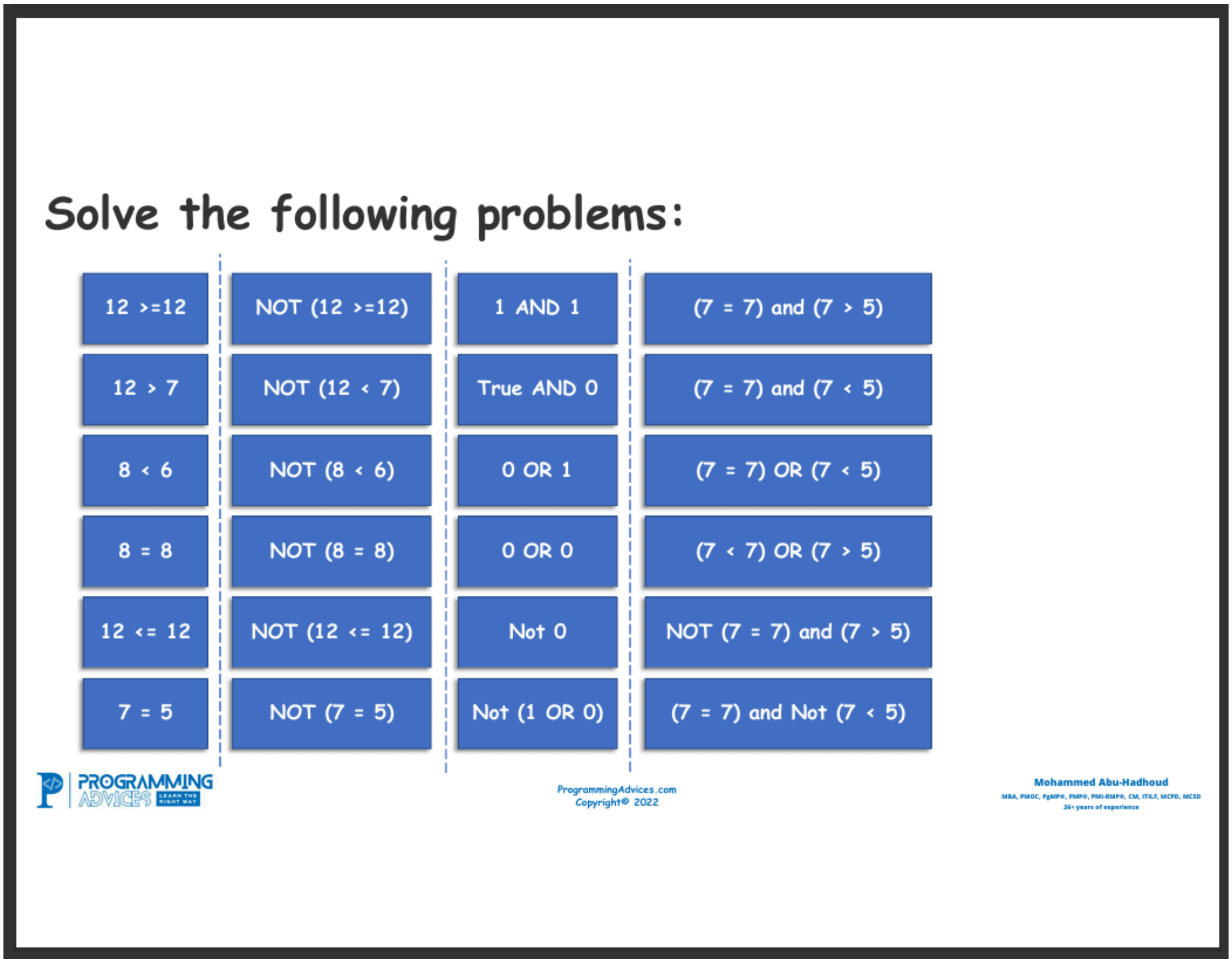
✅ 11. Does a compiler save machine code?

✅ **Yes.**

A compiler **saves the machine code** in a file such as:

- `.exe` (executable)
- `.o` or `.obj` (object files)

This allows you to run the program **without recompiling** every time.



🔷 Column 1: Comparison Operators

1. `12 >= 12` → ✅ **True**
2. `12 > 7` → ✅ **True**
3. `8 < 6` → ❌ **False**
4. `8 = 8` → ✅ **True**
5. `12 <= 12` → ✅ **True**
6. `7 = 5` → ❌ **False**

🔷 Column 2: NOT Operations

1. `NOT (12 >= 12)` → `NOT (True)` → ❌ **False**
2. `NOT (12 < 7)` → `NOT (False)` → ✅ **True**

3. `NOT (8 < 6)` → `NOT (False)` → ✅ **True**
4. `NOT (8 = 8)` → `NOT (True)` → ❌ **False**
5. `NOT (12 <= 12)` → `NOT (True)` → ❌ **False**
6. `NOT (7 = 5)` → `NOT (False)` → ✅ **True**

🔷 Column 3: Boolean Logic (AND/OR/NOT)

1. `1 AND 1` → ✅ **True**
2. `True AND 0` → `True AND False` → ❌ **False**
3. `0 OR 1` → `False OR True` → ✅ **True**
4. `0 OR 0` → ❌ **False**
5. `NOT 0` → `NOT False` → ✅ **True**
6. `NOT (1 OR 0)` → `NOT (True)` → ❌ **False**

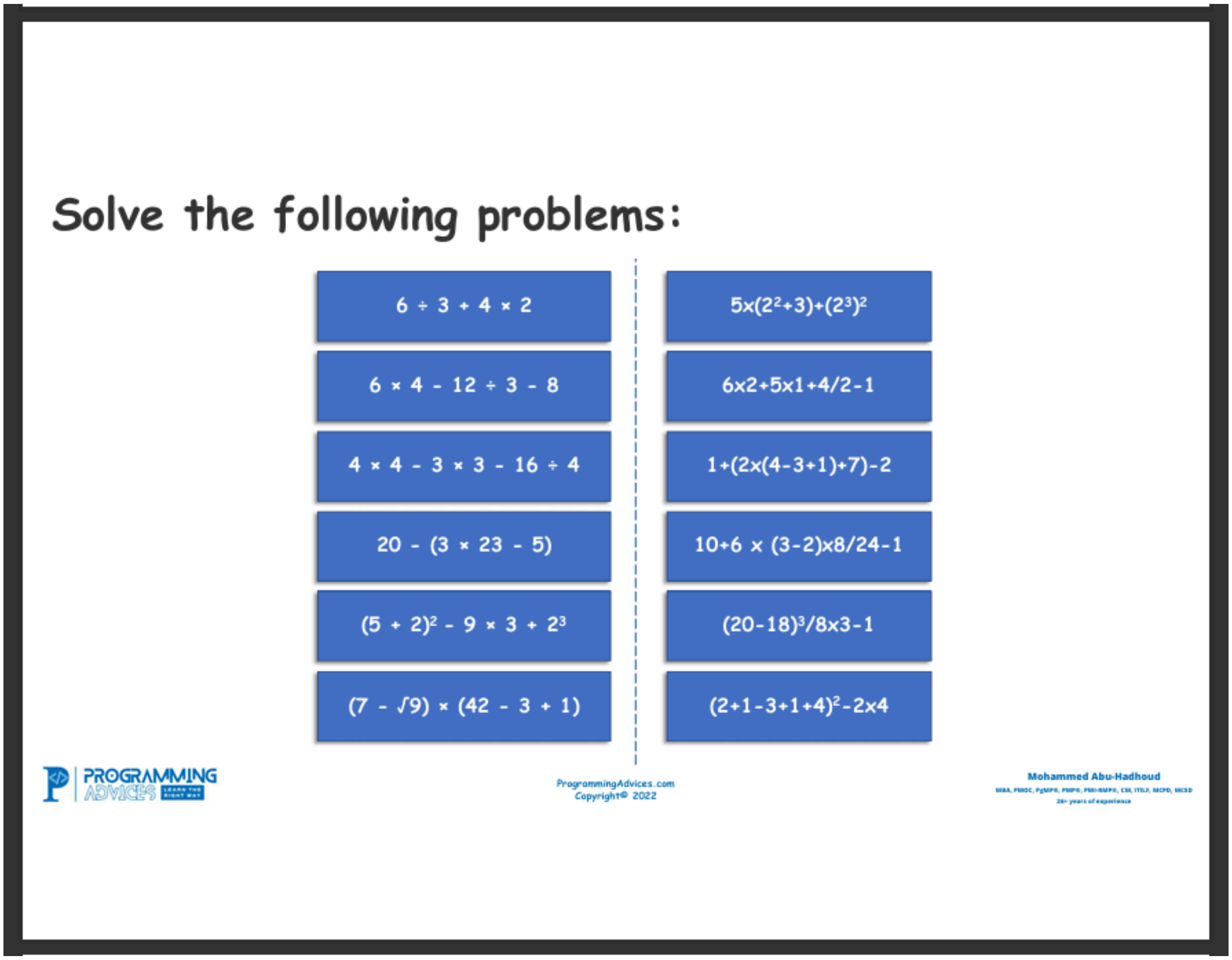
🔷 Column 4: Combined Logic with Comparisons

1. `(7 = 7) and (7 > 5)` → `True and True` → ✅ **True**
2. `(7 = 7) and (7 < 5)` → `True and False` → ❌ **False**
3. `(7 = 7) OR (7 < 5)` → `True OR False` → ✅ **True**
4. `(7 < 7) OR (7 > 5)` → `False OR True` → ✅ **True**
5. `NOT (7 = 7) and (7 > 5)` → `False and True` → ❌ **False**
6. `(7 = 7) and NOT (7 < 5)` → `True and True` → ✅ **True**

✅ Summary Table:

Expression	Result
12 >= 12	True
12 > 7	True
8 < 6	False
8 = 8	True

Expression	Result
12 <= 12	True
7 = 5	False
NOT (12 >= 12)	False
NOT (12 < 7)	True
NOT (8 < 6)	True
NOT (8 = 8)	False
NOT (12 <= 12)	False
NOT (7 = 5)	True
1 AND 1	True
True AND 0	False
0 OR 1	True
0 OR 0	False
NOT 0	True
NOT (1 OR 0)	False
(7 = 7) and (7 > 5)	True
(7 = 7) and (7 < 5)	False
(7 = 7) OR (7 < 5)	True
(7 < 7) OR (7 > 5)	True
NOT (7 = 7) and (7 > 5)	False
(7 = 7) and NOT (7 < 5)	True



🔷 Left Column

1. **6 ÷ 3 + 4 × 2**

   → 2 + 8 = ✅ **10**

2. **6 × 4 - 12 ÷ 3 - 8**

   → 24 - 4 - 8 = ✅ **12**

3. **4 × 4 - 3 × 3 - 16 ÷ 4**

   → 16 - 9 - 4 = ✅ **3**

4. **20 - (3 × 23 - 5)**

   → 20 - (69 - 5) = 20 - 64 = ✅ **-44**

5. **$(5 + 2)^2 - 9 \times 3 + 2^3$**

   → $7^2 - 27 + 8 = 49 - 27 + 8 =$ ✅ **30**

6. **$(7 - \sqrt{9}) \times (42 - 3 + 1)$**

   → $(7 - 3) \times 40 = 4 \times 40 =$ ✅ **160**

🔷 Right Column

1. **$5 \times (2^2 + 3) + (2^3)^2$**

   → $5 \times (4 + 3) + 64 = 5 \times 7 + 64 = 35 + 64 =$ ✅ **99**

2. **$6 \times 2 + 5 \times 1 + 4 \div 2 - 1$**

   → $12 + 5 + 2 - 1 =$ ✅ **18**

3. **$1 + (2 \times (4 - 3 + 1) + 7) - 2$**

   → $1 + (2 \times 2 + 7) - 2 = 1 + 4 + 7 - 2 =$ ✅ **10**

4. **$10 + 6 \times (3 - 2) \times 8 \div 24 - 1$**

   → $10 + 6 \times 1 \times 8 \div 24 - 1$

   → $10 + 48 \div 24 - 1$

   → $10 + 2 - 1 =$ ✅ **11**

5. **$(20 - 18)^3 \div 8 \times 3 - 1$**

   → $2^3 \div 8 \times 3 - 1 = 8 \div 8 \times 3 - 1 = 1 \times 3 - 1 =$ ✅ **2**

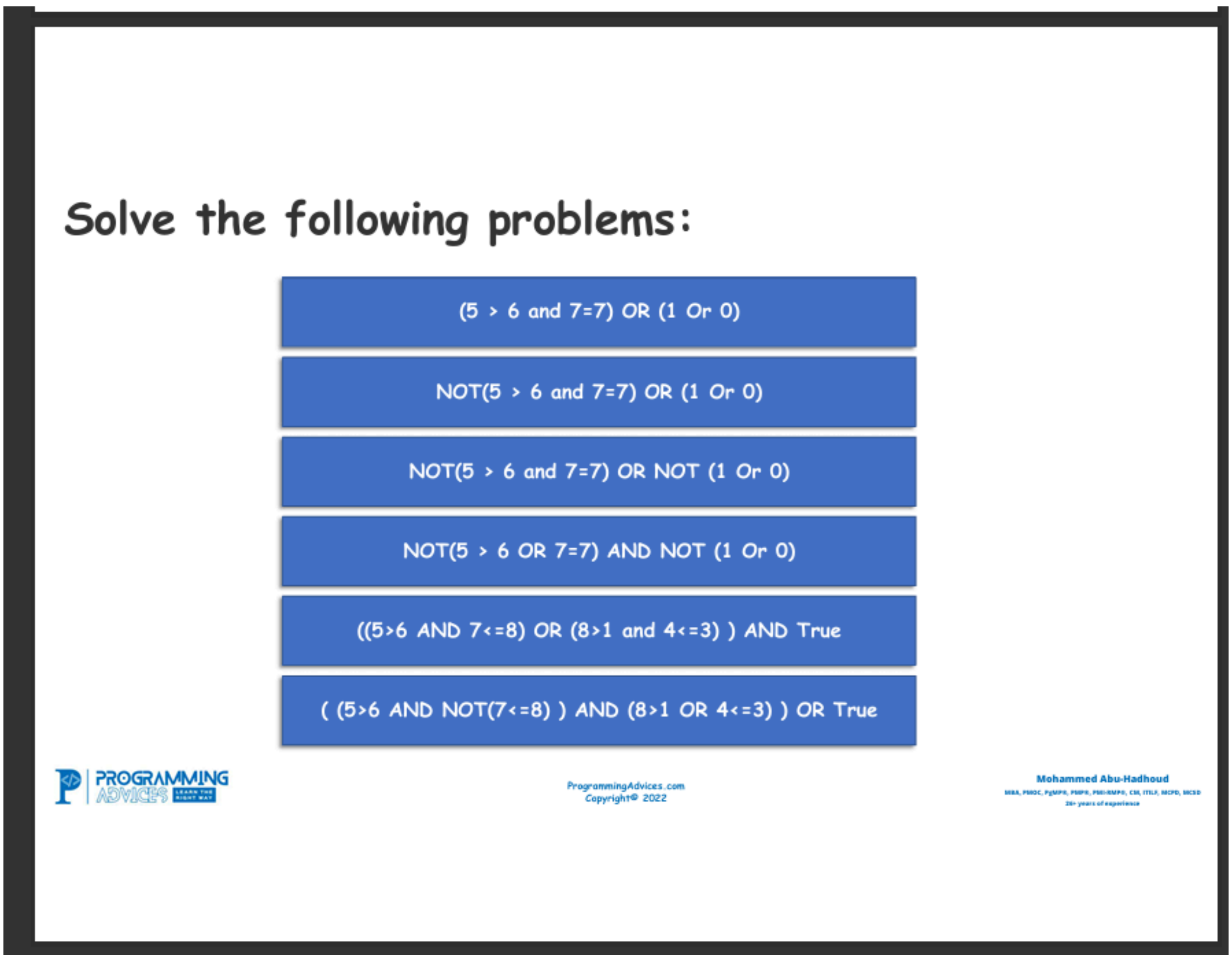
6. **$(2 + 1 - 3 + 1 + 4)^2 - 2 \times 4$**

   → $(5)^2 - 8 = 25 - 8 =$ ✅ **17**

✅ Final Answer Summary:

Expression	Answer
$6 \div 3 + 4 \times 2$	10
$6 \times 4 - 12 \div 3 - 8$	12
$4 \times 4 - 3 \times 3 - 16 \div 4$	3
$20 - (3 \times 23 - 5)$	-44
$(5 + 2)^2 - 9 \times 3 + 2^3$	30
$(7 - \sqrt{9}) \times (42 - 3 + 1)$	160

Expression	Answer
$5 \times (2^2 + 3) + (2^3)^2$	99
$6 \times 2 + 5 \times 1 + 4 \div 2 - 1$	18
$1 + (2 \times (4 - 3 + 1) + 7) - 2$	10
$10 + 6 \times (3 - 2) \times 8 \div 24 - 1$	11
$(20 - 18)^3 \div 8 \times 3 - 1$	2
$(2 + 1 - 3 + 1 + 4)^2 - 2 \times 4$	17



🔷1. (5 > 6 and 7 = 7) OR (1 Or 0)

→ (False and True) OR (True)

→ False OR True

✅ **True**

🔷2. NOT(5 > 6 and 7 = 7) OR (1 Or 0)

→ NOT(False and True) OR (True)

→ NOT(False) OR True

→ True OR True

✅ **True**

🔷3. NOT(5 > 6 and 7 = 7) OR NOT(1 Or 0)

→ NOT(False and True) OR NOT(True)

→ NOT(False) OR False

→ True OR False

✅ **True**

🔷4. NOT(5 > 6 OR 7 = 7) AND NOT(1 Or 0)

→ NOT(False OR True) AND NOT(True)

→ NOT(True) AND False

→ False AND False

✅ **False**

🔷5. ((5 > 6 AND 7 <= 8) OR (8 > 1 AND 4 <= 3)) AND True

→ ((False AND True) OR (True AND False)) AND True

→ (False OR False) AND True

→ False AND True

✅ **False**

🔷6. ((5 > 6 AND NOT(7 <= 8)) AND (8 > 1 OR 4 <= 3)) OR True

→ ((False AND NOT(True)) AND (True OR False)) OR True

→ ((False AND False) AND True) OR True

→ (False AND True) OR True

→ False OR True

✅ **True**

✅ Final Summary:

Expression	Answer
(5 > 6 and 7 = 7) OR (1 Or 0)	**True**
NOT(5 > 6 and 7 = 7) OR (1 Or 0)	**True**
NOT(5 > 6 and 7 = 7) OR NOT(1 Or 0)	**True**
NOT(5 > 6 OR 7 = 7) AND NOT(1 Or 0)	**False**
((5 > 6 AND 7 <= 8) OR (8 > 1 AND 4 <= 3)) AND True	**False**
((5 > 6 AND NOT(7 <= 8)) AND (8 > 1 OR 4 <= 3)) OR True	**True**

1. What is a Variable? And why do we need it?

A **variable** is a symbolic name or identifier used in programming to store data that can change during program execution. It acts as a container for values such as numbers, text, or other data types.

🔷 **Why we need it:**

Variables allow us to store, manipulate, and retrieve values dynamically. They make code more readable, reusable, and adaptable.

✅ **Example:**

```
age = 25
name = "Mustafa"
```

Here, `age` is a variable holding a number, and `name` holds a string.

---

2. What is a Constant? And why do we need it?

A **constant** is similar to a variable, but its value **does not change** during the program's execution.

🔷 **Why we need it:**

Constants help maintain integrity in code by preventing accidental changes to values that should remain fixed (e.g., mathematical constants or configuration settings).

✅ **Example (Python-style):**

```
PI = 3.14159
```

---

3. What is a Memory Cell?

A **memory cell** is the smallest unit of memory in a computer, capable of holding a single piece of data (typically one bit or one byte). Each variable or constant is stored in one or more memory cells.

✅ **Example:**

An `int` value might occupy 4 memory cells (4 bytes).

---

4. What is an Identifier?

An **identifier** is the name used to identify a variable, function, class, module, or object in a program. It must follow certain naming rules depending on the programming language.

✅ **Example:**

```
userName = "Ali"  # "userName" is an identifier
```

---

5. What is a Memory Address? What is its relation to location?

A **memory address** is a unique numerical label assigned to each memory cell, which indicates **where** data is stored in memory.

🔷 **Relation to location:**

The **location** of a variable in memory is determined by its address. The address acts like the street number of a house—it tells the system where to find the data.

✅ **Example:**

If variable `x` is stored at address `0x7ffdf4`, then that's its memory location.

---

6. Which numbering system is used in memory addresses?

Memory addresses are typically represented in the **hexadecimal (base 16)** numbering system, because it's more compact and readable than binary.

✅ **Example:**

`0x7FA2` is a hexadecimal memory address.

---

7. Mention the primary types of variables

The **primary types of variables** (also known as data types) include:

- **Integer (int)** – whole numbers
- **Float (float)** – decimal numbers
- **Character (char)** – single letters
- **String (str)** – sequence of characters
- **Boolean (bool)** – True/False values

---

8. What are the number's types? And give examples

Numerical types are divided mainly into:

- **Integer** – Whole numbers

  ✅ Example: `42`, `-7`

- **Float (or Double)** – Decimal numbers

✅ Example: `3.14`, `-0.001`

- **Unsigned Integer** – Whole numbers without negative values

  ✅ Example: `255` (in many languages like C)

---

9. What is a String? And give an example on it.

A **string** is a sequence of characters used to represent text. It is enclosed in quotes.

✅ **Example:**

```
message = "Hello, World!"
```

---

10. What is Boolean? And give an example on it.

A **Boolean** is a data type that has only two possible values: **True** or **False**. It is commonly used in conditions and logic.

✅ **Example:**

```
isAdult = True
isEmpty = False
```

1. Are all variables and constants the same size?

No, **variables and constants do not have the same size.** The size depends on the **data type** and the architecture of the system (e.g., 32-bit or 64-bit).

🔷 **Explanation:**

Different data types require different amounts of memory:

- `int` might take 4 bytes
- `float` might take 4 or 8 bytes (depending on whether it's a `float` or `double`)
- `char` usually takes 1 byte
- `bool` may take 1 byte (even though it holds just True/False)

✅ **Example (C language):**

```
int a = 10;       // Typically 4 bytes
float b = 3.14f;  // Typically 4 bytes
char c = 'A';     // 1 byte
```

---

2. What happens when you use a much larger size than you need in variables?

When you allocate more memory than needed:

- It results in **wasted memory space**, which may impact performance if done excessively.
- For small programs, it may seem negligible, but in large-scale or embedded systems, it can **cause inefficiency**.

✅ **Example:**

Using a `double` (8 bytes) to store an age value like `25` is inefficient; an `int` (4 bytes) is more appropriate.

---

3. Can we modify a variable during the program?

Yes, **variables are meant to be modified** during program execution. That's their purpose—to hold values that can change.

✅ **Example (Python):**

```
score = 10
score = score + 5  # score is now 15
```

---

4. Can we modify a constant during the program?

No, a **constant cannot be modified** once it is defined. If you try to change it, most programming languages will throw a **compile-time error**.

✅ **Example (C++):**

```
const float PI = 3.14;
PI = 3.1415;  // ❌ Error: cannot modify a constant
```

---

5. How to make a variable read-only?

To make a variable **read-only**, you define it as a **constant** using keywords like `const` (C/C++), `final` (Java), or in some languages, by placing the variable in a protected context.

✅ **Examples:**

- **C++:**

```
const int maxScore = 100;
```

- **Java:**

```
final int maxScore = 100;
```

- **Python (by convention only, not enforced):**

```
MAX_SCORE = 100  # Developers treat ALL_CAPS as constants
```

---

6. Where do the variables and constants get stored?

Variables and constants are stored in different parts of memory:

- **Local variables:** Stored in the **stack**
- **Global/static variables:** Stored in the **data segment**
- **Constants:** Often stored in the **read-only section** of memory
- **Dynamically allocated variables (using malloc/new):** Stored in the **heap**

✅ **Example (C++):**

```
int a = 10;      // stack
const int b = 20;  // read-only segment
```

---

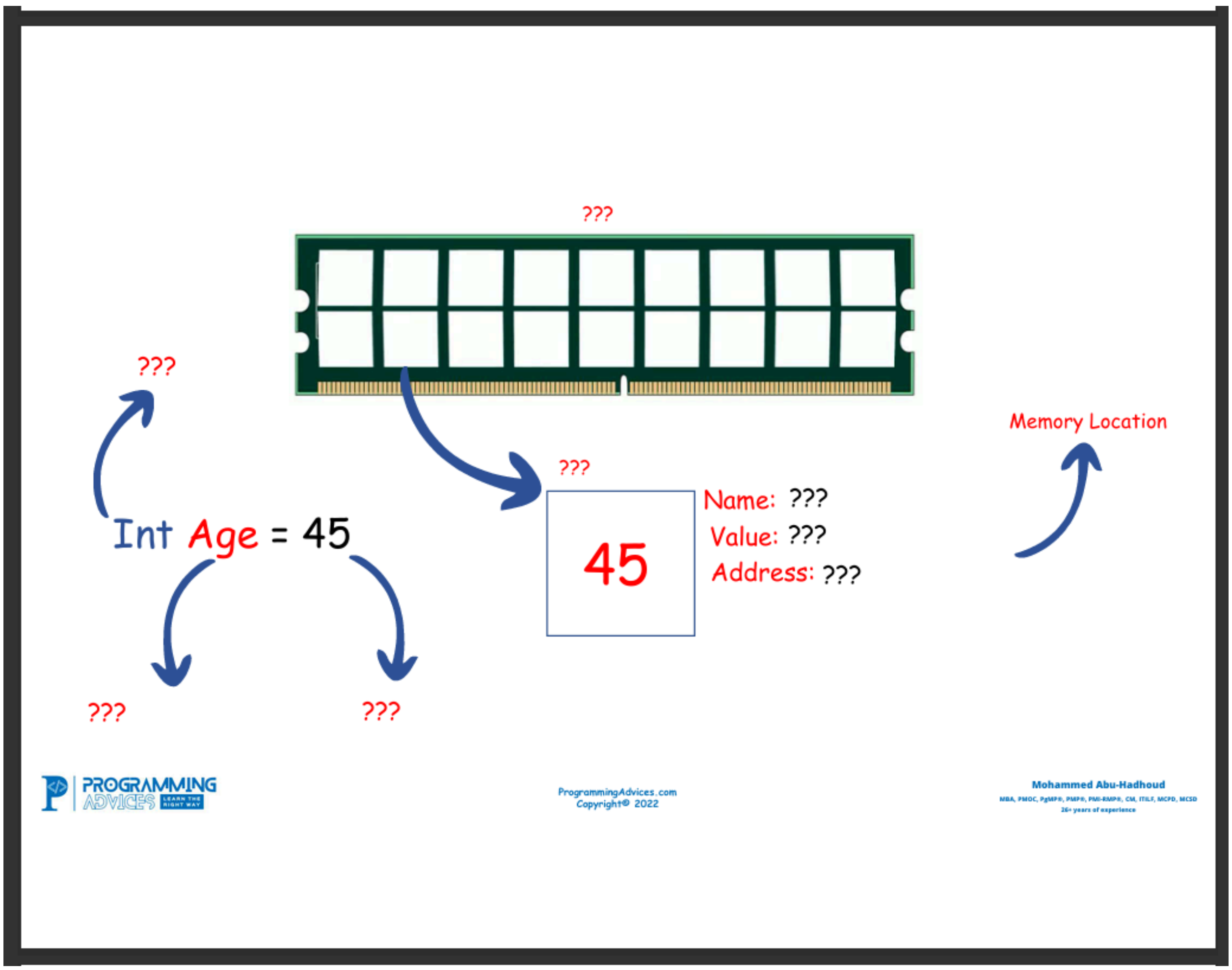
7. What is the difference between Integers and Floats?

Feature	Integer (`int`)	Float (`float`) or Double
Definition	Whole numbers only	Numbers with decimals

Feature	Integer (`int`)	Float (`float`) or Double
Precision	No decimal part	Includes decimal part
Memory usage	Typically 4 bytes	Typically 4–8 bytes
Example	`-5` , `0` , `123`	`3.14` , `-0.001` , `2.0`

✅ **Example:**

```
a = 10     # Integer
b = 10.5   # Float
```



✅ Diagram Explanation: What Happens in Memory When You Write `int Age = 45;`

When you declare a variable like `int Age = 45;`, the compiler reserves a **memory location** to store this data. Let's walk through each part of the diagram and fill in the missing details.

🔷 1. Int Age = 45

- `int` is the **data type**: It tells the system to reserve enough memory to hold an integer, typically **4 bytes**.
- `Age` is the **variable name** (also called the **identifier**): This is how the program refers to the value stored.
- `45` is the **value** assigned to the variable.

🔷 2. What Is Stored in RAM?

The RAM stick shown in the diagram is symbolic. It shows that **some of its cells are used to store program data**, including variables.

- The RAM is divided into **memory cells**.
- Each cell has a **unique address** (like a mailbox number).
- When the line `int Age = 45;` is executed, one of those memory cells is allocated to store the value `45`.

🔷 3. Filling in the Diagram Details:

Label on Diagram	Answer (What to Fill In)
`???` above the RAM stick	**RAM** or **Main Memory** – where variables are stored.
`???` pointing to `int`	**Data Type** – tells the system what kind of value.
`???` pointing to `Age`	**Variable Name / Identifier** – name of the memory cell.
`???` pointing to `45`	**Value Assigned** – the actual data stored.
`???` above the box with `45`	**Memory Cell / RAM Cell** – holds the value.
`Name: ???`	**Age** – variable name.

Label on Diagram	Answer (What to Fill In)
`Value: ???`	**45** – data stored.
`Address: ???`	Example: **0x7ffe12ab** – the memory address (in hex).

🧠 Conceptual Clarification

📌 What is a Variable?

A **variable** is a named reference to a memory location. It allows the programmer to **store, read, and update values** in memory using a readable name (like `Age`), instead of the numeric memory address.

📌 What is a Memory Cell?

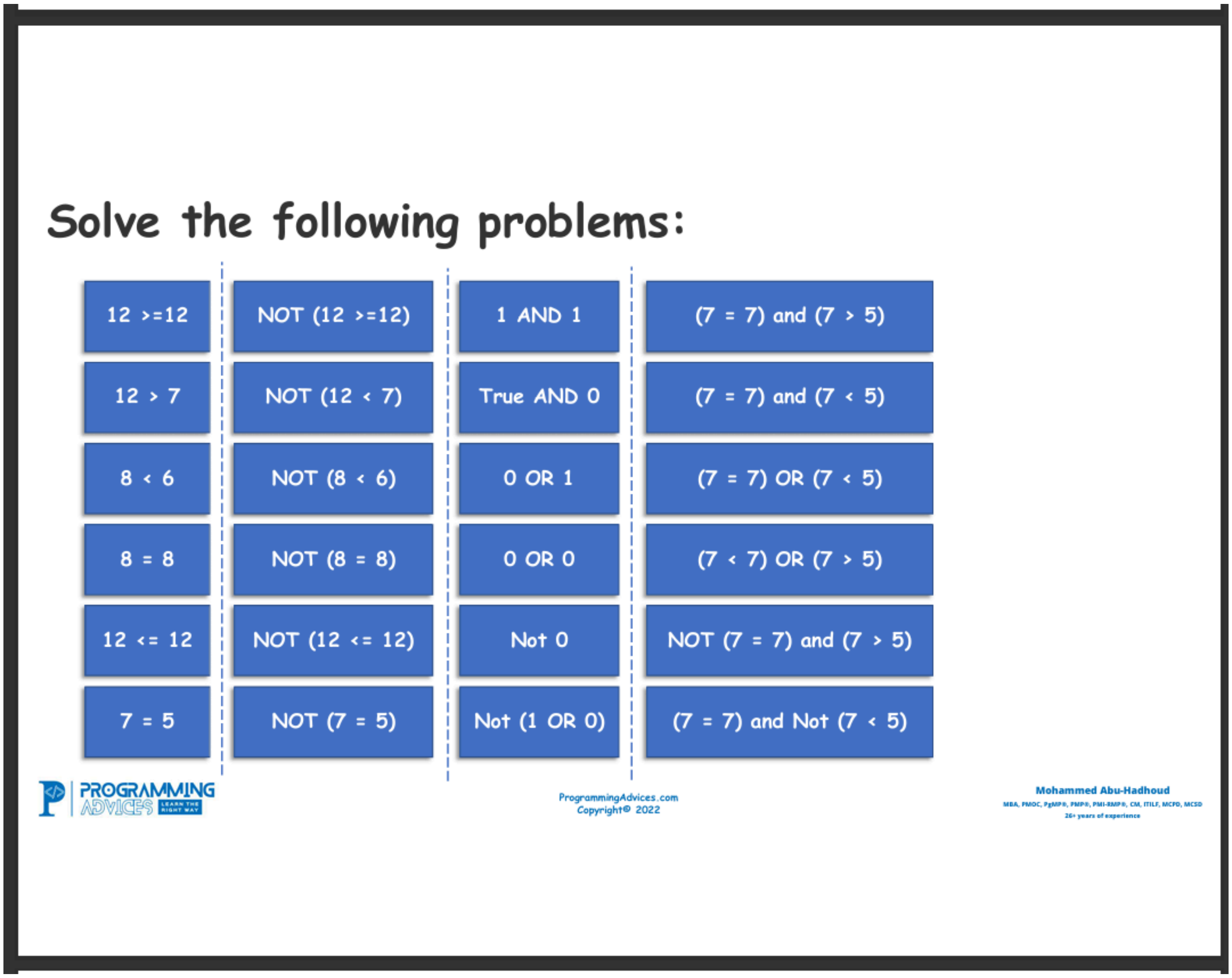
A **memory cell** is the smallest unit of memory that can store data. Each cell has a **fixed size** (usually 1 byte) and a **unique address**. Data types like `int` may require multiple cells (e.g., 4 bytes).

📌 What is a Memory Address?

A **memory address** is a numerical label (like `0x7ffde934`) that uniquely identifies a memory cell. The variable `Age` is **linked to that address** behind the scenes.

✅ Interview Summary Statement

> In short, when we declare int Age = 45;, the compiler tells the operating system to allocate a space in RAM large enough to hold an integer, assigns the value 45 to that space, and maps the variable name Age to the unique memory address of that space. This allows us to access or modify the value using a meaningful name rather than a numerical address, improving both readability and maintainability of the code.



✅ Column 1: Comparison Operators

1. `12 >= 12` → ✅ **True**
2. `12 > 7` → ✅ **True**
3. `8 < 6` → ❌ **False**
4. `8 = 8` → ✅ **True**
5. `12 <= 12` → ✅ **True**
6. `7 = 5` → ❌ **False**

✅ Column 2: NOT Operations

1. `NOT (12 >= 12)` → `NOT (True)` → ❌ **False**

2. `NOT (12 < 7)` → `NOT (False)` → ✅ **True**
3. `NOT (8 < 6)` → `NOT (False)` → ✅ **True**
4. `NOT (8 = 8)` → `NOT (True)` → ❌ **False**
5. `NOT (12 <= 12)` → `NOT (True)` → ❌ **False**
6. `NOT (7 = 5)` → `NOT (False)` → ✅ **True**

✅ Column 3: Logical Operations

1. `1 AND 1` → ✅ **1 (True)**
2. `True AND 0` → ❌ **0 (False)**
3. `0 OR 1` → ✅ **1 (True)**
4. `0 OR 0` → ❌ **0 (False)**
5. `Not 0` → ✅ **True**
6. `Not (1 OR 0)` → `Not (1)` → ❌ **False**

✅ Column 4: Compound Logic Expressions

1. `(7 = 7) and (7 > 5)` → `True and True` → ✅ **True**
2. `(7 = 7) and (7 < 5)` → `True and False` → ❌ **False**
3. `(7 = 7) OR (7 < 5)` → `True OR False` → ✅ **True**
4. `(7 < 7) OR (7 > 5)` → `False OR True` → ✅ **True**
5. `NOT (7 = 7) and (7 > 5)` → `False and True` → ❌ **False**
6. `(7 = 7) and NOT (7 < 5)` → `True and True` → ✅ **True**

✅ Summary Table

Question	Answer
12 >= 12	True
12 > 7	True
8 < 6	False

Question	Answer
8 = 8	True
12 <= 12	True
7 = 5	False
NOT (12 >= 12)	False
NOT (12 < 7)	True
NOT (8 < 6)	True
NOT (8 = 8)	False
NOT (12 <= 12)	False
NOT (7 = 5)	True
1 AND 1	True
True AND 0	False
0 OR 1	True
0 OR 0	False
Not 0	True
Not (1 OR 0)	False
(7 = 7) and (7 > 5)	True
(7 = 7) and (7 < 5)	False
(7 = 7) OR (7 < 5)	True
(7 < 7) OR (7 > 5)	True
NOT (7 = 7) and (7 > 5)	False
(7 = 7) and Not (7 < 5)	True