



# AINPT-IITGN

All India National Proficiency Test - IIT Gandhinagar CDF  
Advanced Intelligence & Aptitude Test

## Comprehensive Test Preparation Guide





# Table of Contents

Module	Topics	Page Number
Programming Fundamentals (Python)	1. Variables & Data Types	3
	2. Python Operators & Expressions	11
	3. Control Flow & Logic	18
	4. Functions, Scope and Modularity	26
	5. Basic Input and Output	34
	6. Collections	41
	7. Tuples	44
Java: String Manipulation, Array/List Operations & Data Structures	1. Essential Concepts & Terminology	48
	2. Problem Solving Methodology	51
	3. Solved Examples	52
	4. Practice Questions	56
SQL Query Fundamentals & Data Retrieval	1. Essential Concepts & Terminology	63
	2. Solved Examples	66
	3. Practice Questions & Self-Assessment	69
Quantitative Reasoning	1. Essential Concepts & Terminology	71
	2. Conceptual Framework & Problem Solving Methodology	76
	3. Solved Examples	80
Data Structures & Algorithms	1. Algorithm Efficiency	85
	2. Data Structures Fundamentals	89
	3. Algorithmic Thinking	92
	4. Problem Decomposition	96
	5. Pseudocode Interpretation	99
	6. Recursion Basics	104
Data Interpretation & Logical Reasoning	1. Essential Concepts & Terminology	107
	2. Conceptual Framework & Problem Solving Methodology	109
	3. Solved Examples	112
	4. Practice Questions & Self-Assessment	115

**\*\*Disclaimer:** This guide covers key concepts and representative question types, but students should be prepared for variations and apply analytical thinking during the actual test. \*\*



# Programming Fundamentals (Python)

## 1: Variables & Data Types

A variable is a named location used to store data in a computer's memory. Python uses variables and their associated data types (numbers, text, or logical values) to process information. Because Python uses dynamic typing, the interpreter automatically determines a variable's type at runtime.

### 1.1 Key Terminology

The following table defines fundamental terms related to variables in Python:

Term	Definition
Variable	A symbolic name referencing a memory location that stores data.
Literal	The raw data value assigned to a variable or constant (e.g., 10, "apple").
Primitive Types	The built-in fundamental data kinds in Python — numbers, booleans, and strings.
Type Casting	The process of converting data from one type to another (e.g., int(), float(), str()).
Scope	Defines where a variable can be accessed within a program.
Constant	A variable whose value is intended not to change, usually written in uppercase (e.g., PI = 3.14).

### 1.2 Primitive Data Types

Python's basic data types fall into three broad categories: **numeric**, **text**, and **boolean**.

Type	Description	Keyword
Integer	Whole numbers (e.g., 1, -5, 76).	int
Float	Numbers with a decimal point (e.g., 1.23, 4.99).	float
String	Sequence of characters enclosed in quotes (' ', " ", or '""').	str
Boolean	Logical truth values, either True or False.	bool

```
students_count = 1000      # int
rating = 4.99              # float
is_published = True        # bool
course_name = "Python 101" # str
two_as_string = "2"       # str
```



## 1.3 Variable Management, Naming, and Literals

### 1.3.1 Variable Declaration and Naming

Variables are created using the **assignment operator (=)**.

#### Concepts

- **Dynamic Typing:** Python automatically infers the data type when a value is assigned.
- **Overwriting:** Assigning a new value to an existing variable replaces the previous one.

#### Good Practices

- Use **meaningful, lowercase names** (e.g., student\_count not sc).
- Separate words with underscores (snake\_case).
- Keep spaces around = for readability (x = 10).
- Avoid using **reserved keywords** (for, class, if, etc.) as variable names.

#### Restrictions

- Cannot start with a digit (1\_name).
- Cannot contain special symbols (!, @, \$, etc.).
- Case-sensitive (name ≠ Name).

#### Examples:

```
# Simple Assignment
age = 18

# Overwriting
name = "Tim"
print(name)    # Output: Tim
name = "Bob"   # Overwrites 'Tim'
print(name)    # Output: Bob

# Multiple Assignments
a, b, c = 5, 3.2, "hello"  # a=5 b=3.2 c="hello"
x = y = z = 100           # Same value for all
```

### 1.3.2 Constants and Literals

**Literals** Literals are raw data or fixed values assigned to variables. Types include **numeric**, **string**, **boolean**, and **special** literals.

Type	Example	Description
Numerical	10, 4.5	Integer or float
String	"Python"	Sequence of characters
Boolean	True, False	Logical truth values
Special	None	Represents "no value" or null

Constants Python doesn't enforce constants syntactically but uses naming conventions.

Use UPPERCASE letters (e.g., PI = 3.14, GRAVITY = 9.8) to indicate immutability.

## 1.4 Type Casting and Scope

### 1.4.1 Type Conversion (Casting)

Type casting is the process of converting a value from one data type to another. Python supports **explicit casting** using built-in functions:



Function	Action	Result Type
int()	Converts to integer (drops decimal part).	int
float()	Converts to float (adds .0 if integer).	float
str()	Converts to string (encloses in quotes).	str

**Common Pitfall:** Attempting to cast a non-numeric string (e.g., int("ABCD")) causes a **ValueError: invalid literal for int()**.

```
x = "10"      # str
y = int(x)    # 10

f = 8.75
i = int(f)    # 8 (int)

i_to_f = float(1) # 1.0 (float)
# invalid_cast = int("Hello") # ValueError
```

### 1.4.2 Variable Scope (LEGB Rule)

Python determines where a variable can be accessed using the **LEGB** hierarchy:

Scope	Definition	Rule / Access
Local	Declared inside a function or block.	Accessible only within that function.
Enclosing	Variables in the outer function of nested functions.	Use nonlocal to modify.
Global	Declared at the top level of a file/module.	Accessible anywhere; use global to modify inside a function.
Built-in	Python's pre-defined names (print, len, min, etc.).	Avoid redefining these names.

#### Analogy

Think of Global scope as a library, Local scope as a study room, and Enclosing scope as a smaller cabin inside. Each room can see the shelves above it, but not below.

### 1.5 Real-World Uses

- **Casting:** Converting input() (always a string) to a number before arithmetic:  
age = int(input("Enter age: "))
- **Booleans:** Used in condition checks: True and False

### 1.6 Solved Problems & Practice Questions

#### Problem 1: Scope Determination



```
GREETING = "Global Message"
def display():
    GREETING = "Local Hi"
    print(GREETING)

display()
print(GREETING)
```

#### Output:

```
Local Hi
Global Message
```

Explanation: The local GREETING inside the function doesn't overwrite the global one.

#### Problem 2: Naming & Syntax

```
name$ = "Alex"
number = 15
int("Six")
```

#### Explanation:

- name\$ → SyntaxError (invalid character \$)
- number = 15 → Valid
- int("Six") → ValueError (invalid literal for int)

#### Problem 3: Type Casting Results

Initial Value	Operation	Variable	Value	Type
a = 10	str(a)	b	"10"	str
c = 5.99	int(c)	d	5	int
e = 1	float(e)	f	1.0	float

#### Sample Questions

1. What data type will x be after: x = False?  
a) String b) Integer **c) Boolean** d) Float
2. What data type results from int("100")?  
a) String **b) Integer** c) Float d) ValueError
3. Assigning 10.5 to a variable gives what type?  
a) Integer b) Decimal **c) Float** d) Complex
4. What happens if you define MIN = 10 globally?  
a) Error b) Auto-lowercase **c) Works but shadows min()** d) Treated as constant
5. Standard naming convention for total\_income?  
a) CamelCase b) Hyphenated **c) Underscore (snake\_case)** d) Space-separated



6. Purpose of global inside a function?  
a) Makes all vars global   b) Declares immutable vars   **c) Modifies a global var**   d) Creates a local var
7. In age = 25, what is 25?  
a) Function   b) Operator   **c) Literal**   d) Constant

### Practice Questions

- What is the type of x after x = False?  
a) str b) int c) bool d) float
- What does float(7) produce?  
a) 7 (int) b) 7.0 (float) c) "7" (str) d) error
- Which is a correct constant by convention?  
a) Limit = 10 b) limit\_value = 10 c) LIMIT\_VALUE = 10 d) limit-value = 10
- Which will raise an error?  
a) int("12") b) float("3.14") c) int("AB12") d) str(5)
- Pick the valid variable name:  
a) total-income b) 2count c) TotalIncome d) total\_income
- Given:

```
x = 5
def g():
    global x
    x = 7
g()
print(x)
```

What prints?

- a) 5 b) 7 c) NameError d) TypeError
7. Which is **not** primitive?  
a) int b) float c) str d) list

## 1.7 Additional Solved Examples

### Solved Example 1 — Dynamic Typing and Overwriting

```
x = 10          # int
print(type(x)) # <class 'int'>
x = "Ten"       # str, overwrites int
print(type(x)) # <class 'str'>
```

  

```
<class 'int'>
<class 'str'>
```



### Explanation:

Python allows the same variable name to reference different types at runtime — this is **dynamic typing**.

### Solved Example 2 — Type Casting Chain

```
a = "3.14"
b = float(a)    # 3.14
c = int(b)      # 3
d = str(c)      # "3"
print(a, b, c, d)
```

```
3.14 3.14 3 3
```

### Concepts Used:

String → Float → Int → String conversions.

### Solved Example 3 — Scope Resolution (LEGB Rule)

```
x = "Global"
def outer():
    x = "Enclosing"
    def inner():
        x = "Local"
        print("Inner:", x)
    inner()
    print("Outer:", x)
outer()
print("Global:", x)
```

```
Inner: Local
Outer: Enclosing
Global: Global
```

### Explanation:

Each function defines its own scope. The variable `x` inside `inner()` doesn't overwrite the one in `outer()` or `global`.

### Solved Example 4 — Boolean Logic

```
a = 10
b = 20
is_greater = a > b
print(is_greater)
```

```
False
```

### Output:

False

### Concept:

Booleans store logical results of comparisons.

### Solved Example 5 — Constant Conventions



```
PI = 3.14159
radius = 7
area = PI * radius ** 2
print(area)
```

```
153.93791
```

### Concept:

Constants are written in uppercase for clarity (though not enforced by Python).

## Complex Practice Problems (with Hints)

### Problem 1 — Mixed Casting & Arithmetic

```
x = "50"
y = 10.5
z = int(x) + y
print(type(z), z)
```

**Question:** What will be printed?

**Hint:** `int("50")` → 50, addition with float promotes result to float.

**Answer:** `<class 'float'> 60.5`

### Problem 2 — Scope Modification

```
x = 5
def update():
    global x
    x = x + 10
update()
print(x)
```

**Question:** Why does this print 15?

**Hint:** The `global` keyword modifies the variable in the global scope.

**Answer:** Output is 15 because `x` is changed globally.

### Problem 3 — Invalid Conversion

```
value = "Python3"
print(int(value))
```

**Question:** What happens and why?

**Hint:** `int()` only works with numeric strings.

**Answer:** `ValueError: invalid literal for int()`.

### Problem 4 — Nested Scope & Nonlocal



```
def outer():  
    msg = "Outer Message"  
    def inner():  
        nonlocal msg  
        msg = "Inner Modified"  
    inner()  
    print(msg)  
outer()
```

**Question:** What prints?

**Answer:** Inner Modified

**Explanation:** nonlocal allows modifying the enclosing function's variable.

### Problem 5 — Evaluate Expressions Dynamically

```
a = "10"  
b = "20"  
sum_result = int(a) + int(b)  
concat_result = a + b  
print(sum_result, concat_result)
```

**Output:**

30 1020

**Concept:**

Type conversion changes meaning — arithmetic vs. string concatenation.

### Problem 6 — Boolean Evaluation Challenge

```
val = ""  
num = 0  
check = bool(val) or bool(num)  
print(check)
```

**Output:**

False

**Explanation:**

Empty string "" and 0 both evaluate to False.

## 1.8 Advanced Challenge Problems

### 1. Type Guessing

```
x = str(5 * 2.0)  
print(type(x))
```

What is the data type of x? Why?



## 2. LEGB Conflict

```
x = "Global"
def func():
    x = "Local"
    def inner():
        print(x)
    inner()
func()
```

Predict output and explain the scope resolution order.

## 3. Implicit Casting

```
a = 3
b = 4.0
print(a * b)
```

Why is the result a float?

## 4. Global Masking

```
len = 10
print(len(["a", "b"]))
```

What error occurs? Why should built-in names not be redefined?

## 5. Comprehensive Type Flow

```
a = "100.5"
b = float(a)
c = int(b)
d = str(c)
print(a, b, c, d, sep=" | ")
```

Predict each variable's type and value.

# 2. Python Operators and Expressions

This chapter provides a review of arithmetic, comparison, logical, and assignment operators.

## 2.1 Arithmetic Operators

These operators are used to perform standard mathematical operations.

Operator	Name	Example	Result
+	Addition: adds two operands	5 + 3	8



Operator	Name	Example	Result
-	Subtraction: subtracts two operands	5 - 3	2
*	Multiplication: multiplies two operands	5 * 3	15
/	<b>True Division:</b> divides the first operand by the second	5 / 2	2.5
//	<b>Floor Division:</b> divides the first operand by the second, removes the decimal	5 // 2	2
%	Modulo: returns the remainder when the first operand is divided by the second	5 % 2	1
**	Exponentiation: Returns first raised to power second	5 ** 2	25

- **True vs. Floor Division:** True division (/) *always* returns a float, even if the result is a whole number (e.g., 10 / 5 is 2.0). Floor division (//) truncates the decimal portion, returning an int if both operands are ints.
- **Modulo (%):** This returns the remainder of a division. It is extremely useful for checking for divisibility (e.g., num % 2 == 0 checks if num is even)

## 2.2 Comparison (Relational) Operators

These operators compare two values and return a Boolean (True or False) result.

Operator	Name	Example	Result
==	Equal to	a == b	True if a and b are equal
!=	Not equal to	a != b	True if a and b are not equal
>	Greater than	a > b	True if a is greater than b
<	Less than	a < b	True if a is less than b
>=	Greater than or equal to	a >= b	True if a is greater than or equal to b
<=	Less than or equal to	a <= b	True if a is less than or equal to b

Do not confuse the assignment operator (=), which *assigns* a value, with the comparison operator (==), which *checks* for equality. This is a very common source of bugs.

## 2.3 Logical Operators

These operators are used to combine or invert Boolean expressions (True/False).



Operator	Name	Example	Result
and	Logical AND	$x > 5$ and $x < 10$	True if <i>both</i> conditions are true
or	Logical OR	$x > 5$ or $y > 5$	True if <i>at least one</i> condition is true
not	Logical NOT	not(is_admin)	True if the condition is False (inverts)

## 2.4 Assignment and Compound Operators

These operators are used to assign values to variables.

Operator	Example	Equivalent to
=	$x = 5$	$x = 5$
+=	$x += 3$	$x = x + 3$
-=	$x -= 3$	$x = x - 3$
*=	$x *= 3$	$x = x * 3$
/=	$x /= 3$	$x = x / 3$
%=	$x %= 3$	$x = x \% 3$
//=	$x //= 3$	$x = x // 3$
**=	$x **= 3$	$x = x ** 3$

Augmented assignment (like  $x +=$ ) can be more efficient for mutable objects (like lists) as it often modifies the object *in-place*.

## 2.5 Operator Precedence (The "Vertical" Order)

Precedence defines the order of operations, similar to PEMDAS (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction) in mathematics. Operators with higher precedence are evaluated before operators with lower precedence.

**Example:**  $\text{result} = 10 + 20 * 30$  Because  $*$  has higher precedence than  $+$ , the multiplication  $20 * 30$  (600) is performed first. The expression then becomes  $10 + 600$ , resulting in 610

### Simplified Precedence Table (Highest to Lowest):

Precedence	Operator	Description
Highest	()	Parentheses (overrides all precedence)



Precedence	Operator	Description
High	**	Exponentiation
	*\$, \$/\$, \$//\$, \$\%\$	Multiplication, Division, Floor, Modulo
	+\$, -\$	Addition, Subtraction
	==, !=, >, <, >=, <=	Comparison Operators
	not	Logical NOT
	and	Logical AND
Lowest	or	Logical OR

## 2.6 Operator Associativity (The "Horizontal" Order)

Associativity determines the evaluation order for operators that have the *same* precedence.

- **Left-to-Right Associativity:** Most Python operators are left-associative. **Example:**  $100 / 10 * 10$   
Since  $/$  and  $*$  have the same precedence, left-associativity applies.
  1. The expression is evaluated as  $(100 / 10) * 10$ .
  2.  $10.0 * 10$
  3. Result:  $100.0$  This is *not* evaluated as  $100 / (10 * 10)$ , which would be  $1.0$ .
- **Right-to-Left Associativity:** The main exception is the exponentiation operator  $**$ , which is right-associative. **Example:**  $2 ** 2 ** 3$ 
  1. The expression is evaluated as  $2 ** (2 ** 3)$ .
  2.  $2 ** 8$
  3. Result:  $256$  This is *not* evaluated as  $(2 ** 2) ** 3$ , which would be  $4 ** 3 = 64$ .

## 3. Using Parentheses ()

Parentheses have the highest precedence and can be used to explicitly control and override the order of operations.

- $10 - 4 * 2 = 2$  (Precedence)
- $(10 - 4) * 2 = 12$  (Parentheses force a different order)

**Good Practice:** When in doubt, use parentheses. They make your code more readable and prevent subtle precedence-related bugs.

## 2.7 Solved Problems & Practice Questions

### Solved Problems

#### **Problem 1:** Precedence Evaluation

Expression:  $y = 5 + 3 * 2 ** 2$

What is the value of  $y$ ?

Explanation:

1. **Exponentiation ( $**$ )** has the highest precedence:  $2 ** 2 = 4$ .
2. **Multiplication ( $*$ )** is next:  $3 * 4 = 12$ .
3. **Addition ( $+$ )** is last:  $5 + 12 = 17$ .

Output: 17



### Problem 2: Associativity Evaluation

Expression:  $z = 5 - 2 + 3$

What is the value of  $z$ ?

Explanation:

- and + have the same precedence. They are left-associative.

1. The expression is evaluated as  $(5 - 2) + 3$ .
2.  $3 + 3 = 6$ .

Output: 6

### Problem 3: Logical Operator Precedence

Expression:  $\text{result} = \text{False or True and not False}$

What is the value of result?

Explanation:

1. **not** has the highest logical precedence: not False is True.
2. **and** is next: True and True (from step 1) is True.
3. **or** is last: False or True (from step 2) is True.

Output: True

### Sample Questions

1. What is the data type of the result of  $10 / 4$ ?  
a) int b) float c) str d) bool
2. Which operator has the highest precedence in the following list?  
a) + b) and c) \*\* d) ==
3. What is the output of  $\$10 // 3$ ?  
a) 3.333 b) 3 c) 1 d) 3.0
4. What is the output of  $\$2 ** 3 ** 2$ ?  
a) 64 b) 12 c) 512 d) Error
5. What will  $x$  be after  $\$x = 5; x *= 3$ ?  
a) 3 b) 5 c) 15 d) 8
6. The expression  $A \text{ or } B$  short-circuits and does not evaluate  $B$  if:  
a)  $A$  is True b)  $A$  is False c)  $B$  is True d)  $B$  is False

Answers: 1(b), 2(c), 3(b), 4(c), 5(c), 6(a)

### Additional Solved Examples

#### Solved Example 1 — Mixed Division

```
a = 15
b = 4
true_div = a / b
floor_div = a // b
modulo = a % b
print(true_div, floor_div, modulo)
```

```
3.75 3 3
```

Concept: Demonstrates the three different outcomes of division operations.



## Solved Example 2 — Short-Circuiting Safety

```
count = 0
total = 100
# This check is safe
if count > 0 and total / count > 50:
    print("Average is high")
else:
    print("Cannot calculate or average is low")

# This would cause an error
# if total / count > 50 and count > 0:
```

Cannot calculate or average is low

Concept: The and operator short-circuited. Because `count > 0` was False, the `total / count` expression (which would cause a `ZeroDivisionError`) was never executed.

## Complex Practice Problems (with Hints)

Problem 1 — Complex Expression Evaluation

Expression: `result = (10 > 5 and 3 < 1) or not (5 == 5)`

Question: What will be printed?

Hint: Evaluate the expressions inside the parentheses first. Remember the precedence: not, then and, then or.

Answer: False

Explanation:

1. `(10 > 5 and 3 < 1) -> (True and False) -> False`
2. `not (5 == 5) -> not (True) -> False`
3. `False or False -> False`

Problem 2 — Right-Associativity Challenge

Expression: `result = 4 ** 3 ** 2 / (2 ** 2)`

Question: What is the type and value of result?

Hint: `**` is right-associative. `/` (True Division) always produces a float.

Answer: `<class 'float'> 65536.0`

Explanation:

1. `4 ** 3 ** 2 -> 4 ** (3 ** 2) -> 4 ** 9 -> 262144`
2. `(2 ** 2) -> 4`
3. `262144 / 4 -> 65536.0`

## Advanced Challenge Problems

Problem 1 — Truthy Values and Logic

```
x = 0
y = "Hello"
z = ""
result = x or y or z
print(result)
```

Question: Predict the output.

Hint: Logical operators return the value that determines the outcome, not just True or False. 0, "", and `` are "Falsy". Non-empty strings/lists are "Truthy".



Answer: Hello

Explanation: x (0) is False. Python moves to y ("Hello"), which is True. Due to short-circuiting, y's value is returned immediately, and z is never checked.

Problem 2 — Chained Comparisons

```
print(1 < 2 < 3)
print(1 < 3 > 2)
```

Question: What do these two lines print?

Hint: Python allows comparison "chaining." It's syntactic sugar.

Answer:

True

True

Explanation:

1.  $1 < 2 < 3$  is evaluated as  $(1 < 2)$  and  $(2 < 3) \rightarrow \text{True}$  and  $\text{True} \rightarrow \text{True}$ .
  2.  $1 < 3 > 2$  is evaluated as  $(1 < 3)$  and  $(3 > 2) \rightarrow \text{True}$  and  $\text{True} \rightarrow \text{True}$ .
-



## 3. Control Flow and Logic

### 3.1 Conditional Branching

#### 3.1.1 The if Statement

The if statement is the most basic control flow statement. It checks a single condition. If the condition evaluates to True, the indented block of code (the "suite") is executed. If the condition is False, the block is skipped, and execution continues after the if block

```
age = 20
if age >= 18:
    print("You are eligible to vote.")
print("Execution continues here.")
```

```
You are eligible to vote.
Execution continues here.
```

#### 3.1.2 The if-else Statement

The if-else statement provides an alternative path of execution.

- if: Runs its block if the condition is True.
- else: Runs its block *only if* the if condition evaluates to False

```
num = -5
if num >= 0:
    print("The number is positive or zero.")
else:
    print("The number is negative.")
```

```
The number is negative.
```

#### 3.1.3 The if-elif-else Chain

This construct is used to check multiple, mutually exclusive conditions in a sequence.

- if: Checks the first condition.
- elif: Short for "else if." This checks a new condition only if all preceding if and elif conditions were False. You can have zero or more elif parts.
- else: This is an optional final block. It runs only if all preceding if and elif conditions have evaluated to False.



```
score = 85

if score >= 90:
    print("Grade: A")
elif score >= 80: # Only checked if score < 90
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else: # Runs if score < 70
    print("Grade: F")
```

Grade: B

- The if...elif...else chain is a sequence of *mutually exclusive* checks. As soon as one condition is met, its block is executed, and the rest of the chain is skipped.

## 3.2 Iteration (Loops)

### 3.2.1 for Loops

The Python for loop is a "for-each" loop. It iterates over the items of any **sequence** (like a list or a string) in the order they appear.

```
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w)
```

cat  
window  
defenestrate

**Counting Loops with range():** To perform a "counting" loop (e.g., "run this 5 times"), the range() function is used.

- range(5) generates numbers 0, 1, 2, 3, 4.
- range(1, 6) generates 1, 2, 3, 4, 5.
- range(0, 10, 2) generates 0, 2, 4, 6, 8.
- 

```
# A standard "counting" loop
for i in range(3):
    print(i)
```

0  
1  
2



### 3.2.2 while Loops

A while loop repeats a block *as long as* a specified condition remains True. The condition is checked *before* each iteration (it is an **entry-controlled** loop).

```
countdown = 3
while countdown > 0:
    print(countdown)
    countdown -= 1 # Must update the variable
print("Liftoff!")
```

```
3
2
1
Liftoff!
```

If the condition variable is not updated within the loop (e.g., if `countdown -= 1` was forgotten), the loop will run forever (an **infinite loop**).

### 3.3.3. Break, Continue, and Pass Statements

These statements alter the flow of a loop from within.

- **break Statement:** The break statement terminates the *nearest enclosing loop* immediately. Execution transfers to the first statement *after* the loop. Use Case: Searching for an item. Once the item is found, the loop can stop

```
for i in range(10):
    if i == 3:
        break # Exits the loop when i is 3
    print(i)
```

```
0
1
2
```

- **continue Statement:** The continue statement skips the rest of the code *in the current iteration* and moves directly to the *next* iteration of the loop. Use Case: Skipping certain items in



a sequence.

```
for i in range(10):  
    if i % 2 == 0: # If i is even  
        continue # Skip to the next iteration  
    print(i)      # This line only runs for odd numbers
```

```
1  
3  
5  
7  
9
```

- **pass Statement:** The pass statement is a null operation; it does nothing. It is used as a placeholder where syntax requires a statement, but no code needs to be executed (e.g., in an empty function or if block). It does not affect loop flow.

```
if i == 3:  
    pass # Placeholder for future code
```

### 3.4 Solved Problems & Practice Questions

#### Solved Problems

**Problem 1:** break vs. continue Code:

```
for num in range(1, 8):  
    if num % 3 == 0:  
        break  
    if num % 2 == 0:  
        continue  
    print(num)
```

```
1
```

What is the output?

**Explanation:**

1. num = 1: Not divisible by 3. Not divisible by 2. print(1).
2. num = 2: Not divisible by 3. Is divisible by 2. continue (skips print).
3. num = 3: Is divisible by 3. break (terminates loop).

Output: 1

**Problem 2:** Nested Loops and break Code:



```
for i in range(3):
    for j in range(3):
        if i == 1 and j == 1:
            break
        print(f"i={i}, j={j}")
    print("----")
```

```
i=0, j=0
i=0, j=1
i=0, j=2
---
i=1, j=0
---
i=2, j=0
i=2, j=1
i=2, j=2
---
```

What is the output?

**Explanation:** The break statement only exits the *inner* loop (the j loop).

1. i=0: Inner loop runs fully (j=0, j=1, j=2). Prints "----".
2. i=1: Inner loop runs for j=0. When j=1, break hits and the inner loop terminates. Prints "----".
3. i=2: Inner loop runs fully (j=0, j=1, j=2). Prints "----". *Output:* i=0, j=0 i=0, j=1 i=0, j=2 --- i=1, j=0 --- i=2, j=0 i=2, j=1 i=2, j=2 ---

### Sample Questions

1. Which statement immediately exits the *entire* loop?  
a) pass b) continue c) exit d) break
2. Which statement skips the *current iteration* and moves to the next?  
a) pass b) continue c) skip d) break
3. What is the output of for i in range(1, 4): print(i)?  
a) 1 2 3 4 b) 0 1 2 3 c) 1 2 3 d) 0 1 2
4. A while loop is an:  
a) Entry-controlled loop b) Exit-controlled loop c) do-while loop d) for loop
5. To emulate a do-while loop, you would typically use: a) for i in range() b) while True: with a break c) if-elif-else d) match...case

*Answers:* 1(d), 2(b), 3(c), 4(a), 5(b)

### Additional Solved Examples

**Solved Example 1 — for loop with else**



```
# A 'for-else' loop
for i in range(1, 5):
    print(i)
    if i == 10:
        break
else:
    # This block runs ONLY if the loop
    # completes without a 'break'.
    print("Loop finished normally.")
```

```
1
2
3
4
Loop finished normally.
```

Concept: The else block on a loop executes if the loop terminates *naturally* (i.e., not via a break statement). This is useful for search loops to check if an item was found.

**Solved Example 2 — Basic Pattern Printing Task:** Print a 4x4 square of asterisks.

```
size = 4
for i in range(size):
    for j in range(size):
        print('*', end=' ')
    print() # Move to the next line
```

```
* * * *
* * * *
* * * *
* * * *
```

Concept: Nested loops are the standard pattern for working with 2D structures or patterns. The outer loop controls the rows; the inner loop controls the columns.

### Complex Practice Problems (with Hints)

#### Problem 1 — Finding a Prime Number

**Task:** Write a script to check if the number  $n = 13$  is prime. A prime number is only divisible by 1 and itself.

**Hint:** Use a for loop to iterate from 2 up to  $n-1$ . If  $n$  is divisible by *any* number in that range (i.e.,  $n \% i == 0$ ), it is not prime. Use a break and a for-else block.



Answer:

```
n = 13
is_prime = True
for i in range(2, n):
    if n % i == 0:
        is_prime = False
        break # Found a factor, no need to check more

if is_prime:
    print(f"{n} is prime.")
else:
    print(f"{n} is not prime.")

13 is prime.
```

(Alternative using for-else):

```
n = 13
for i in range(2, n):
    if n % i == 0:
        print(f"{n} is not prime.")
        break
else:
    # This runs if the loop never 'broke'
    print(f"{n} is prime.")

13 is prime.
```

### Advanced Challenge Problems

**Problem 1 — Loop Variable Scope** Code:

```
i = 5
for i in range(3):
    print(i)
print(i)
```

**Question:** What is the output of the *final* print(i)?

**Hint:** In Python, the loop variable is *not* local to the loop. It "leaks" into the surrounding scope and overwrites any existing variable with the same name.

**Answer:** 2

**Explanation:** The for loop rebinds the variable i. After the loop finishes, i holds the *last value* it was assigned, which was 2. The original i = 5 is overwritten.

**Problem 2 — while-else** Code:



```
i = 1
while i < 5:
    print(i)
    i += 1
    if i == 3:
        pass # Does nothing
else:
    print("While loop else block")
```

*Question:* What is the output?

*Hint:* Like for-else, the while-else block runs only if the loop terminates normally (i.e., its condition becomes False), not via a break.

*Answer:* 1 2 3 4 While loop else block

*Explanation:* The loop condition  $i < 5$  becomes False when  $i$  is 5. The loop terminates normally, so the else block is executed

---



## 4. Functions, Scope and Modularity

Functions are blocks of code designed to perform a specific task. They are defined once and can be "called" (invoked) repeatedly from other parts of the program. This principle of *modularity* makes code easier to read, test, and maintain.

The following table defines key function concepts:

Term	Definition
<b>Definition (def)</b>	The process of creating a function using the <code>def</code> keyword, giving it a name, parameters, and a code block (suite).
<b>Call (Invoke)</b>	The act of <i>executing</i> a function by using its name followed by parentheses (e.g., <code>my_function()</code> ).
<b>Parameter</b>	A <i>variable name</i> listed in a function's definition. It acts as a placeholder for an input value (e.g., <code>name</code> in <code>def hi(name):</code> ).
<b>Argument</b>	The <i>actual value</i> or object that is passed to a function when it is called (e.g., <code>"Alice"</code> in <code>hi("Alice")</code> ).
<b>Scope (LEGB)</b>	The rule Python uses to determine the visibility of a variable, searching in the <b>L</b> ocal, <b>E</b> nclosing, <b>G</b> lobal, and <b>B</b> uilt-in namespaces.

### 4.1 Function Definition and Calling

- **Definition Syntax:** Uses the `def` keyword, a function name, parentheses for parameters, and a colon. The indented block below is the function's "body."

```
def greet(name):  
    print("Hello", name)
```

- **Calling Syntax:** Uses the function name followed by parentheses containing the required arguments.

```
greet("World") # 'World' is the argument
```

```
Hello World
```

### 4.2 Parameters and Arguments (Pass-by-Object-Reference)

The query's "pass by value concept" is a common simplification, but Python's mechanism is more nuanced. Python is **pass-by-object-reference** (also called pass-by-assignment).



1. When a function is called, Python passes the *reference* (memory address) of the argument object.
2. The function's *parameter* becomes a new name (alias) for that *exact same object*.
3. What happens next depends on whether the object is **immutable** or **mutable**.
- **Immutable Arguments** (int, str, float, tuple): These objects *cannot* be changed. Any attempt to "change" them inside the function only reassigns the local parameter to a *new* object. The caller's original variable is unaffected. This *behaves like* pass-by-value.

```
def change_string(s):
    s = "new value" # Reassigns 's' to a new string object
    print(f"Inside: {s}")

message = "old value"
change_string(message)
print(f"Outside: {message}")

Inside: new value
Outside: old value
```

**Mutable Arguments** (list, dict, set): These objects *can* be changed in-place.

- **Case 1: Mutating the object.** If you use methods that change the object (e.g., `.append()`, `.pop()`), the changes *will* be visible to the caller, as both variables point to the same list. This *behaves like* pass-by-reference.
- **Case 2: Reassigning the parameter.** If you assign the parameter to a new object (e.g., `my_list =`), you are only changing the *local* variable. This breaks the link to the original object, and the caller is unaffected.

```
def change_list(data):
    # Case 1: Mutation (affects caller)
    data.append('a')

    # Case 2: Reassignment (does not affect caller)
    data =
    print(f"Inside (after reassign): {data}")

my_numbers =
change_list(my_numbers)
print(f"Outside: {my_numbers}")
```

*Output:* Inside (after reassign): Outside: [1, 2, 3, 'a'] (The `.append('a')` mutation persisted; the reassignment did not)



## 4.3 Return Values and Types

- **The return Statement:** The return keyword exits a function and sends a value (or object) back to the code that called it.
- **print vs. return:** This is a critical distinction.
  - **print():** Displays a value to the console for human viewing. print() itself is a function that returns None.
  - **return:** Passes a value back to the program to be stored in a variable or used in another expression.

```
def add(a, b):  
    return a + b # Returns the result to the caller  
  
sum_value = add(5, 10) # 'sum_value' becomes 15  
# print(sum_value)
```

- **Implicit None Return:** A function that does not have a return statement (or has a bare return) automatically returns the special value None.
- **Returning Multiple Values:** Python functions can return multiple values by separating them with commas. Python automatically *packs* these values into a single tuple. The caller can then *unpack* the tuple into multiple variables.

```
def get_user_info():  
    name = "Alice"  
    age = 30  
    return name, age # Packs into ('Alice', 30)  
  
# Unpacking the returned tuple  
user_name, user_age = get_user_info()  
  
print(f"Name: {user_name}, Age: {user_age}")
```

Output: Name: Alice, Age: 30

## 4.4 Scope and Modularity

### 4.4.1. Function Scope (LEGB Rule)

Scope defines where a variable can be accessed. Python uses the **LEGB** rule to resolve variable names, searching namespaces in this order :

1. **L (Local):** Names defined *inside* the current function (including parameters). A new local scope is created for every function call.
2. **E (Enclosing):** Names defined in the local scope of any *enclosing* functions (used for nested functions).
3. **G (Global):** Names defined at the top level of the script/module, or explicitly declared with the global keyword.
4. **B (Built-in):** Python's pre-defined names (e.g., print(), len(), str()).



```
x = "Global" # G

def outer():
    x = "Enclosing" # E
    def inner():
        x = "Local" # L
        print(f"Inner sees: {x}")

    inner()
    print(f"Outer sees: {x}")

outer()
print(f"Global sees: {x}")
# print(len("test")) # 'len' is found in Built-in (B)
```

Inner sees: Local  
Outer sees: Enclosing  
Global sees: Global

- **global Keyword:** To *modify* a global variable from *inside* a function, you must explicitly state your intention with the global keyword.

```
count = 0
def increment():
    global count
    count += 1
```

- **nonlocal Keyword:** To *modify* an enclosing (E) variable from an inner (L) function, you must use the nonlocal keyword.

## 4.5 Built-in Functions

Python provides many built-in functions that are always available (in the 'B' scope).

- `len(iterable)`: Returns the number of items (length) in a sequence or collection.
- `max(iterable)`: Returns the largest item in an iterable.
- `min(iterable)`: Returns the smallest item in an iterable.
- `sum(iterable)`: Returns the sum of all items in an iterable (which must be numeric).

## 4.6 Solved Problems & Practice Questions

### Solved Problems

#### Problem 1: Scope Determination



```
GREETING = "Global Message"
def display():
    GREETING = "Local Hi"
    print(GREETING)
display()
print(GREETING)
```

```
Local Hi
Global Message
```

*Explanation:* The assignment `GREETING = "Local Hi"` creates a *new* local variable inside `display()`. It does not affect the global variable of the same name.

**Problem 2: Modifying with global Code:**

```
x = 5
def g():
    global x
    x = 7
g()
print(x)
```

```
7
```

*Explanation:* The `global x` keyword tells the function `g()` to modify the *global* variable `x`, not create a new local one. The change persists.

**Problem 3: Mutable Parameter Puzzle Code:**

```
def update(data):
    data.append('new')
    data = '' # Reassignment
    return data

my_data = ['old']
returned_val = update(my_data)

print(my_data)
print(returned_val)
```

```
['old', 'new']
```

*What is printed?*

**Explanation:**

1. `my_data` (a list) is passed by object reference to `update()`. `data` and `my_data` point to the *same* list `['old']`.
2. `data.append('new')`: This *mutates* the shared list. `my_data` is now `['old', 'new']`.
3. `data = ''`: This *reassigns* the *local* variable `data` to a *new* list. It breaks the link to `my_data`.
4. `return data`: The function returns the *new* list. \*Output:\* ``['old', 'new']``

**Sample Questions**



1. In `def my_func(x);`, `x` is a(n):  
a) Argument b) Parameter c) Return value d) Global variable
2. In `my_func(10)`, `10` is a(n):  
a) Argument b) Parameter c) Return type d) Local variable
3. A function with no explicit return statement returns:  
a) 0 b) True c) None d) Error
4. To modify a variable from an *enclosing* (not global) scope, you must use:  
a) global b) enclosing c) outer d) nonlocal
5. If you pass a list to a function and reassign it (e.g., `my_list =` ) inside, does the caller's list change?  
a) Yes, always b) No, reassignment is local c) Only if the list is empty
6. Which of these functions returns the number of items in a list?  
a) `max()` b) `sum()` c) `len()` d) `count()`

Answers: 1(b), 2(a), 3(c), 4(d), 5(b), 6(c)

### Additional Solved Examples

#### Solved Example 1 — print vs. return

```
def func_print(x):  
    print(x) # Just displays to console  
  
def func_return(x):  
    return x # Gives the value back  
  
a = func_print(10)  
b = func_return(10)  
  
print(f"a is: {a}")  
print(f"b is: {b}")  
  
10  
a is: None  
b is: 10
```

Concept: `func_print` printed "10" but returned None to `a`. `func_return` returned 10 to `b`.

#### Solved Example 2 — Unpacking Multiple Return Values

```
def analyze_list(numbers):  
    total = sum(numbers)  
    count = len(numbers)  
    average = total / count  
    return total, count, average # Returns a tuple  
  
# Unpacking the tuple  
t, c, avg = analyze_list([10,20,30])  
print(f"Total: {t}, Average: {avg}")  
  
Total: 60, Average: 20.0
```

Concept: Returning multiple values as a tuple is a common and clean Python pattern.



## Complex Practice Problems (with Hints)

### Problem 1 — Palindrome Checker

*Task:* Write a function `is_palindrome(s)` that takes a string `s` and returns `True` if it's a palindrome (reads the same forwards and backward) and `False` otherwise.

*Hint:* Use `return` with a Boolean comparison. A string's reverse can be found using `s[::-1]`. *Answer:*

```
def is_palindrome(s):  
    return s == s[::-1]  
  
print(is_palindrome("racecar"))  
print(is_palindrome("hello"))
```

```
True  
False
```

### Problem 2 — Nested Scope and nonlocal Code:

```
def outer():  
    msg = "Outer Message"  
    def inner():  
        nonlocal msg  
        msg = "Inner Modified"  
    inner()  
    print(msg)  
outer()
```

```
Inner Modified
```

*Question:* What prints?

*Hint:* `nonlocal` allows `inner()` to modify the `msg` variable from the `outer()` scope.

## Advanced Challenge Problems

### Problem 1 — Default Argument Mutable Pitfall Code:

Python

```
def add_to_list(item, my_list=[]):  
    my_list.append(item)  
    return my_list  
  
print(add_to_list(1))  
print(add_to_list(2))  
print(add_to_list(3))
```

```
[1]  
[1, 2]  
[1, 2, 3]
```

*Question:* Predict the output.

*Hint:* Default arguments are created *once*, when the function is *defined*. If a default argument is *mutable* (like a list), it is shared across all calls.



*Explanation:* All three calls are appending to the *same* list, which was created in memory when `def add_to_list` was first run. The "Pythonic" way to avoid this is to use `my_list=None` as the default and create a new list inside the function.

### Problem 2 — LEGB Conflict Code:

```
x = "Global"
def func():
    x = "Local"
    def inner():
        print(x)
    inner()
func()
```

Local

*Question:* Predict the output and explain the scope resolution. ]

*Explanation:* `inner()` needs to find `x`. It searches its own **Local** scope (empty). It searches its **Enclosing** scope (the scope of `func()`). It finds `x = "Local"` in `func()`'s scope and stops searching. The **Global** `x` is never seen.

---



## 5. Basic Input and Output

I/O (Input/Output) is the mechanism by which a program communicates with the outside world (e.g., a user). The `input()` function *pauses* the program's execution to wait for user-typed data.<sup>37</sup> The `print()` function sends data to the standard output (usually the console).

A fundamental concept in Python I/O is that `input()` *always* returns data as a string. This necessitates explicit type conversion (casting) before the data can be used in numerical calculations.<sup>38</sup>

The following table defines key I/O concepts:

Term	Definition
<b>input()</b>	A built-in function that reads a line from the user, converts it to a string, and returns it.
<b>print()</b>	A built-in function that displays one or more values to the console.
<b>Type Casting</b>	The process of explicitly converting a value from one data type to another (e.g., <code>int()</code> , <code>float()</code> , <code>str()</code> ).
<b>String Formatting</b>	The process of embedding variables, literals, and expressions inside a string (e.g., f-strings).

### 5.1. `input()` Function and Type Casting

- **The Golden Rule:** The `input()` function *always* returns a string, regardless of what the user types.

```
age = input("Enter your age: ") # User types 25
print(type(age))

Enter your age: 25
<class 'str'>
```

- **The Consequence (Type Casting):** To perform mathematical operations, you *must* cast the string to a numeric type (like `int` or `float`) using the casting functions.

```
age = input("Enter your age: ")
age_int = int(age)

# Or, more commonly, nested:
age = int(input("Enter your age: "))
print(f"Next year, you will be {age + 1}")

Enter your age: 25
Enter your age: 25
Next year, you will be 26
```

- **The Pitfall (ValueError):** If the user enters a non-numeric string (e.g., "ten"), attempting to cast it with `int()` will raise a `ValueError` and crash the program.

```
invalid_cast = int("Hello") # Raises ValueError
```

(Robust programs use `try-except` blocks to handle this, which is a more advanced topic).



## 5.2 Handling Multiple Inputs

It is common to ask for multiple values on a single line. This is achieved by combining `input()` with the `.split()` string method.

1. `input()` reads the *entire line* as a single string (e.g., "10 20 30").
  2. `.split()` splits that string into a *list of strings* (e.g., ['10', '20', '30']).
  3. This list can then be unpacked or processed.
- **Unpacking into variables:**

```
# User inputs: 5 10
x, y = input("Enter two numbers: ").split()

print(x, y)
print(type(x))
# Pitfall: x and y are still strings!
print(int(x) + int(y)) # Must cast
```

```
Enter two numbers: 2 3
2 3
<class 'str'>
5
```

- `map()` Function (Common Idiom): To cast all items in the split list at once, the `map()` function is often used.

```
# User inputs: 2 3 4
x, y, z = map(int, input("Enter three numbers: ").split())
print(x + y + z)
```

```
Enter three numbers: 2 3 4
9
```

- List Comprehension (Alternative): A list comprehension can achieve a similar result, especially if storing in a list.

```
# User inputs: 1 2 3 4 5
numbers = [int(i) for i in input("Enter numbers: ").split()]
print(sum(numbers))
```

```
Enter numbers: 1 2 3 4 5
15
```



## 5.3 Printing Output

### 5.3.1 print() Function Basics

The `print()` function can take multiple arguments, which it will display separated by a space by default.

`print("Hello", "World")` # Output: Hello World

It also has optional parameters:

- `sep=`: Specifies the separator to use *between* items (default is a space).
- `end=`: Specifies what to print at the *end* (default is a newline, `\n`).

```
print("apple", "banana", "cherry", sep=", ")
print("First line", end=" | ")
print("Second line")
```

```
apple, banana, cherry
First line | Second line
```

### 5.3.2 String Formatting Basics (f-strings)

While Python has several ways to format strings (e.g., % formatting, `.format()`), the modern, preferred, and most readable method (since Python 3.6) is using **f-strings**.

- **Syntax:** Prefix the string with an `f` or `F`. Place variables or expressions inside curly braces `{}`.

```
name = "Alice"
age = 30
print(f"Hello, my name is {name} and I am {age} years old.")
```

```
Hello, my name is Alice and I am 30 years old.
```

- **Expressions:** You can put any valid Python expression inside the braces.

```
a = 10
b = 20
print(f"The sum of {a} and {b} is {a + b}.")
```

```
The sum of 10 and 20 is 30.
```

- **Format Specifiers:** f-strings also allow for powerful formatting by adding a colon (`:`) after the expression.
  - o `:.2f`: Formats a float to 2 decimal places.
  - o `:<10`: Left-aligns text within 10 spaces.



- o :%B %d, %Y: Formats a date object.

```
pi = 3.14159265
print(f"The value of pi is approximately {pi:.2f}.")
```

```
The value of pi is approximately 3.14.
```

- Printing Braces: To print literal {} characters, double them: {{}}.

```
print(f"This shows literal braces: {{}}")
```

```
This shows literal braces: {}
```

## 5.4 Solved Problems & Practice Questions

### Solved Problems

**Problem 1: The input() String Trap Code:**

```
a = input('Enter first number: ')
b = input('Enter second number: ')
ans = a + b
print(ans)
```

```
Enter first number: 10
Enter second number: 20
1020
```

*Scenario:* The user enters 10 and then 20. What is the output?

*Explanation:* input() returns strings. a is "10" and b is "20". The + operator on two strings performs *string concatenation*, not mathematical addition.

### Problem 2: Fixing the String Trap

*Task:* How do you fix Problem 1 to print the mathematical sum?

*Answer:* Cast the inputs to int().

```
a = int(input('Enter first number: '))
b = int(input('Enter second number: '))
ans = a + b
print(ans)
```

```
Enter first number: 10
Enter second number: 20
30
```

*Scenario:* The user enters 10 and then 20.



### Problem 3: Formatting Floats

**Task:** Given price = 49.99 and tax = 0.075, calculate the total and print it formatted as a currency (2 decimal places).

**Answer:**

```
price = 49.99
tax = 0.075
total = price * (1 + tax)
print(f"Total price: ${total:.2f}")
```

```
Total price: $53.74
```

### Sample Questions

1. The input() function always returns what data type?  
a) int b) str c) float d) depends on what the user types
2. What character prefixes a string to make it an f-string?  
a) \$ b) & c) % d) f
3. What is the output of print("A", "B", "C", sep="-")?  
a) A B C b) ABC c) A-B-C d) A\nB\nC
4. Which line will cause a ValueError if the user types "five"?  
a) x = input("Enter: ") b) x = "five" c) print(x) d) x = int(input("Enter: "))
5. How do you print a literal { and } in an f-string?  
a) \{ and \} b) "{ and }" c) {{ and }} d) f"{"
6. Which line reads two space-separated *integers* from one line into x and y?  
a) x, y = input().split() b) x, y = int(input().split()) c) x, y = map(int, input().split()) d) x, y = int(input())

**Answers:** 1(b), 2(d), 3(c), 4(d), 5(c), 6(c)

### Additional Solved Examples

#### Solved Example 1 — Reading and Splitting

**Task:** Read three names from a single input line and print them.

```
# User inputs: Alice Bob Charlie
names_line = input("Enter three names: ")
name1, name2, name3 = names_line.split()

print(f"First: {name1}, Second: {name2}, Third: {name3}")
```

```
Enter three names: Alice Bob Charlie
First: Alice, Second: Bob, Third: Charlie
```

**Concept:** .split() with no arguments splits on any whitespace.

#### Solved Example 2 — Splitting by a Different Delimiter

**Task:** Read comma-separated-values (CSV) and print the list.



```
# User inputs: 10,20,30,40
csv_line = input("Enter values separated by commas: ")
values = csv_line.split(',')
print(values)
```

```
Enter values separated by commas: 10,20,30,40
['10', '20', '30', '40']
```

*Concept:* The `.split()` method can take an argument to specify the delimiter string.

### Complex Practice Problems (with Hints)

#### Problem 1 — Simple Calculator

*Task:* Ask the user for two numbers (on *separate* lines). Print their sum, difference, product, and quotient.

*Hint:* Remember to cast both inputs to `float()` to handle decimals and avoid `ValueErrors` from non-integer input (like 10.5).

*Answer:*

```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

print(f"{num1} + {num2} = {num1 + num2}")
print(f"{num1} - {num2} = {num1 - num2}")
print(f"{num1} * {num2} = {num1 * num2}")
print(f"{num1} / {num2} = {num1 / num2}")
```

#### Problem 2 — Name and Age Formatter

*Task:* Ask the user for their name and age *in a single line*. Then print a formatted message.

*Hint:* Use `.split()` and unpack into two variables. Remember to cast age but not name.

*Answer:*

```
# User inputs: Bob 42
name, age_str = input("Enter your name and age: ").split()
age = int(age_str) # Cast only the age

print(f"Welcome, {name}. You are {age} years old.")
```

```
Enter your name and age: Bob 42
Welcome, Bob. You are 42 years old.
```

### Advanced Challenge Problems

#### Problem 1 — Columnar Data Printing

*Task:* Given two lists, `products =` and `prices = [1.2, 0.5, 0.75]`, loop through them and print them in neat, 10-character-wide columns.

*Hint:* Use an f-string format specifier. `<10` left-aligns in 10 spaces, and `>10` right-aligns.



Answer:

```
products = ['Apple', 'Banana', 'Orange']
prices = [1.2, 0.5, 0.75]

print(f"{'Product':<10} | {'Price':>10}")
print("-" * 23)
for i in range(len(products)):
    product = products[i]
    price = prices[i]
    print(f"{'product':<10} | ${price:>10.2f}")
```

Product		Price
Apple	\$	1.20
Banana	\$	0.50
Orange	\$	0.75

## Problem 2 — Summing Comma-Separated Numbers

**Task:** Ask the user for a single line of comma-separated numbers (e.g., "5,10,15"). Calculate and print their sum.

**Hint:** Use `input().split(',')` and a list comprehension (or `map`) to cast each string to an `int`. Then use the built-in `sum()` function.

Answer:

```
# User inputs: 5,10,15,20
line = input("Enter comma-separated numbers: ")
numbers = [int(n) for n in line.split(',')]
total = sum(numbers)
print(f"Total: {total}")
```

```
Enter comma-separated numbers: 5,10,15,20
Total: 50
```



## 6. Collections

This chapter provides a conceptual refresher on Python's sequence collections: Lists and Tuple. The focus is *not* on their full methods, but on the core mechanics of *accessing* data. This includes 0-based indexing, negative indexing, and slicing, which are common to all sequence types.

### 6.1. What is a List?

A **list** is a collection of items that is:

- **Ordered**: The items have a specific order.
- **Changeable** (or mutable): We can add, remove, or modify items in a list.
- **Heterogeneous**: A list can store different types of data (e.g., numbers, strings, other lists).

#### 6.1.1 Creating a List

Lists are created using square brackets [], with each item separated by a comma.

```
my_list = [10, "apple", 3.14, "banana"]
```

In this example:

- 10 is a number (integer),
- "apple" and "banana" are words (strings),
- 3.14 is a decimal number (float).

This shows that lists can hold different data types, which is why they are called **heterogeneous**.

#### 6.1.2 What is Indexing?

**Indexing** helps us access individual items in a list. In Python, items in a list are numbered, starting from **0** for the first item.

```
my_list = ["red", "blue", "green", "yellow"]
```

Index	Item
0	"red"
1	"blue"
2	"green"
3	"yellow"

To get a specific item, we use the list name followed by the index in square brackets.

```
print(my_list[0]) # Output: "red"
print(my_list[2]) # Output: "green"
```

#### Negative Indexing

In Python, you can also use **negative indexing** to access items from the end of the list. Here's how it works with my\_list above:



Negative Index	Item
-1	"yellow"
-2	"green"
-3	"blue"
-4	"red"

```
print(my_list[-1]) # Output: "yellow"
print(my_list[-2]) # Output: "green"
```

### 6.1.3. Basic List Operations

Here are some common operations you can do with lists.

#### Adding Items to a List

1. **append**: Adds a **single** item to the end of the list.

```
fruits = ["apple", "banana"]
fruits.append("orange")
print(fruits) # Output: ["apple", "banana", "orange"]
```

2. **extend**: Adds multiple items (from another list) to the end of the list.

```
fruits = ["apple", "banana"]
more_fruits = ["grape", "melon"]
fruits.extend(more_fruits)
print(fruits) # Output: ["apple", "banana", "grape", "melon"]
```

3. **insert**: Adds an item at a specific index in the list.

```
fruits = ["apple", "banana"]
fruits.insert(1, "orange")
print(fruits) # Output: ["apple", "orange", "banana"]
```

#### Removing Items from a List

1. **pop**: Removes an item by index (default is the last item if no index is given).

```
fruits = ["apple", "banana", "cherry"]
fruits.pop() # Removes "cherry"
print(fruits) # Output: ["apple", "banana"]
```

2. **remove**: Removes a specific item by value.



```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
print(fruits) # Output: ["apple", "cherry"]
```

```
numbers = [10, 20, 30, 40, 50]

# Pop
numbers.pop(2)
print(numbers) # Output: [10, 20, 40, 50]

# Remove
numbers.remove(40)
print(numbers) # Output: [10, 20, 50]
```

#### 6.1.4. Slicing a List

**Slicing** lets us select a part (or "slice") of a list. It's like taking a smaller portion of the list based on index positions.

##### Syntax for Slicing

```
list[start:end]
```

- **start:** The index to start the slice (inclusive).
- **end:** The index to stop the slice (exclusive).

```
numbers = [10, 20, 30, 40, 50]
print(numbers[1:4]) # Output: [20, 30, 40]
```

##### Slicing with Default Values

- **Omitting start:** Starts from the beginning of the list.
- **Omitting end:** Goes to the end of the list.

```
print(numbers[:3]) # Output: [10, 20, 30]
print(numbers[2:]) # Output: [30, 40, 50]
```

##### Reversing a List or Tuple with Slicing

You can use slicing to create a reversed version of a list or tuple without changing the original.

Example: Reversing a List with Slicing



```
numbers = [1, 2, 3, 4, 5]
reversed_numbers = numbers[::-1]
print(reversed_numbers) # Output: [5, 4, 3, 2, 1]
```

### 6.1.5. Useful Functions with Lists

**Sum:** The **sum** function adds up all numeric items in a list.

```
numbers = [1, 2, 3, 4, 5]
print(sum(numbers)) # Output: 15
```

**max and min:** **max** finds the largest item in a list, **min** finds the smallest item in a list.

```
numbers = [1, 2, 3, 4, 5]
print(max(numbers)) # Output: 5
print(min(numbers)) # Output: 1
```

**Concatenation (+):** Lists can be **combined** using the + operator.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined_list = list1 + list2
print(combined_list) # Output: [1, 2, 3, 4, 5, 6]
```

---

## 7. Tuples

### 7.1 What is a Tuple?

A **tuple** is similar to a list but has a key difference: **tuples are immutable**, meaning we **cannot change** them once created. Tuples are useful when you want a fixed collection of items.

#### 7.1.1 Creating a Tuple

Tuples are created using parentheses () instead of square brackets [].

```
my_tuple = (10, "apple", 3.14, "banana")
```

#### 7.1.2 Accessing Tuple Items with Indexing

Just like lists, tuples support **indexing** and **negative indexing**.



```
print(my_tuple[0]) # Output: 10
print(my_tuple[-1]) # Output: "banana"
```

## 7.3 Solved Problems & Practice Questions

### Solved Problems

#### Problem 1: Indexing Review Code:

```
s = "Refresher"
print(s)
print(s[-1])
print(s)
```

```
Refresher
r
Refresher
```

*Explanation:* s is the 2nd char. s[-1] is the last char. s is the 5th char.

#### Problem 2: Slicing Review Code:

```
s = "Refresher"
print(s[2:5])
print(s[5:])
print(s[:4])
```

```
fre
sher
Refr
```

*Explanation:*

- s[2:5]: Indices 2, 3, 4 ('f', 'r', 'e').
- s[5:]: Indices 5 to the end ('s', 'h', 'e', 'r').
- s[:4]: Indices 0 to 3 ('R', 'e', 'f', 'r').

#### Problem 3: Advanced Slicing (Step) Code:

```
s = "abcdefghij"
print(s[1:8:2])
print(s[::-1])
```

```
bdfh
jihgfedcba
```

*Explanation:*

- s[1:8:2]: Start at 1 ('b'). Stop *before* 8. Step by 2. (Indices: 1, 3, 5, 7).
- s[::-1]: Default start and stop, with a step of -1. Reverses the string.



## Sample Questions

1. In Python, the index of the *first* element in a list is:  
a) 1 b) 0 c) -1 d) depends on the list
2. In `my_list =` , what is `my_list[-2]`?  
a) 10 b) 20 c) 30 d) 40
3. What is the index of the *last* element of a string `s`?  
a) `len(s)` b) `len(s) - 1` c) 0 d) 99
4. The expression `"Hello"[1:4]` evaluates to:  
a) "ell" b) "ello" c) "Hel" d) "el"
5. What is the output of `"Python"[:-1]`?  
a) "Python" b) "nohtyP" c) "Pto" d) Error
6. The primary difference between a list and a string is that a list is:  
a) 0-indexed b) Sliced with `[:]` c) Immutable d) Mutable

Answers: 1(b), 2(c), 3(b), 4(a), 5(b), 6(d)

## Additional Solved Examples

### Solved Example 1 — `len()` and Last Item

Task: Access the last item of a list, two ways.

```
my_list = [1,2,3,4]

# Method 1 (C-style, verbose)
last_index = len(my_list) - 1
print(my_list[last_index])

# Method 2 (Pythonic, preferred)
print(my_list[-1])
```

```
4
4
```

Concept: Negative indexing is a more direct and readable way to access items from the end of a sequence.

**Solved Example 2 — Slicing Gracefully** Task: What happens when you slice with an index that is "out of bounds"?

```
word = "Python"
print(word[1:100])
print(word[99:100])
```

```
ython
```

Concept: Slicing does *not* raise an `IndexError`. It gracefully handles out-of-bounds indices by stopping at the end (or start) of the string. Indexing (e.g., `word`), however, *will* raise an `IndexError`.

## Complex Practice Problems (with Hints)

### Problem 1 — Extracting with Step

Task: Given `s = "abcdefghij"`, write a slice to get the string "hj".



*Hint:* You need to start at 'h' (index 7) and take every 2nd character.

*Answer:* s[7::2]

*Explanation:* Starts at index 7 ('h'). No stop is given, so it goes to the end. step is 2. It takes index 7 ('h') and index 9 ('j').

## Problem 2 — Slicing and Immutability

*Task:* "Fix" a string s = "Jython" to be "Python".

*Hint:* Strings are immutable. You cannot assign to s. You must build a *new* string using concatenation and slicing.

*Answer:*

```
s = "Jython"
s = 'P' + s[1:]
print(s)
```

Python

---



# Java: String Manipulation, Array/List Operations & Data Structures

## 1. Essential Concepts & Terminology

### 1.1 String Manipulation Concepts

#### String Concatenation

**Definition:** Combining two or more strings into a single string.

**Syntax:**

```
String result = string1 + string2;  
String result = string1.concat(string2);
```

**Example:**

```
String firstName = "John";  
String lastName = "Doe";  
String fullName = firstName + " " + lastName; // "John Doe"
```

**Key Points:**

- Use + operator for simple concatenation
- Use concat() method for method chaining
- Remember to add spaces manually when needed

#### String Formatting

**Definition:** Creating formatted strings using placeholders and format specifiers.

**Syntax:**

```
String result = String.format("format string", arguments);
```



## Format Specifiers:

SPECIFIER	TYPE	EXAMPLE
%s	String	String.format("Hello %s", "World") → "Hello World"
%d	Integer	String.format("Age: %d", 25) → "Age: 25"
%f	Float/Double	String.format("Price: %.2f", 19.99) → "Price: 19.99"
%c	Character	String.format("Grade: %c", 'A') → "Grade: A"

## Essential String Methods Summary

METHOD	DESCRIPTION	EXAMPLE
length()	Returns the number of characters	"Hello".length() → 5
toUpperCase()	Converts to uppercase	"hello".toUpperCase() → "HELLO"
toLowerCase()	Converts to lowercase	"HELLO".toLowerCase() → "hello"
trim()	Removes leading/trailing spaces	" hi ".trim() → "hi"
substring(start, end)	Extracts part of string	"Hello".substring(0,2) → "He"
equals(str)	Compares two strings	"hi".equals("hi") → true
charAt(index)	Gets character at position	"Hello".charAt(0) → 'H'
replaceAll(old, new)	Replaces all occurrences	"aa".replaceAll("a", "b") → "bb"

## 1.2 Array Concepts

### Array Characteristics

- **Fixed Size:** Size must be declared at creation and cannot change
- **Homogeneous:** All elements must be of the same data type
- **Indexed Access:** Access elements using index (starts at 0)
- **Mutable:** Elements can be modified after creation

### Important: Zero-Based Indexing

Arrays in Java use zero-based indexing. For an array of size 5:



- First element: array[0]
- Last element: array[4] (NOT array[5])
- Valid indices: 0, 1, 2, 3, 4

### Array Declaration & Operations

OPERATION	SYNTAX	EXAMPLE
Declaration with Size	type[] name = new type[size];	int[] numbers = new int[5];
Declaration with Initializer	type[] name = {values};	int[] numbers = {1, 2, 3, 4, 5};
Get Element	name[index]	int x = numbers[0]; // Gets first element
Set Element	name[index] = value;	numbers[0] = 10; // Sets first element
Get Size	name.length	int size = numbers.length;

## 1.3 ArrayList Concepts

### ArrayList Characteristics

- **Dynamic Size:** Can grow or shrink automatically as needed
- **Objects Only:** Stores objects, not primitives (use Integer, not int)
- **Easy Operations:** Built-in methods for add, remove, search
- **Flexible:** More versatile than arrays for dynamic data

### ArrayList Core Operations

OPERATION	ARRAY	ARRAYLIST
Import	Not needed	import java.util.ArrayList;
Declaration	int[] arr = new int[5];	ArrayList<Integer> list = new ArrayList<>();
Add Element	arr[0] = 10;	list.add(10);
Get Element	int x = arr[0];	int x = list.get(0);
Set Element	arr[0] = 20;	list.set(0, 20);
Remove Element	Not directly possible	list.remove(0);
Get Size	arr.length	list.size()



## 2. Problem-Solving Methodology

### 2.1 String Problem-Solving Framework

#### Step 1: Identify Required Operations

Determine which string methods are needed (concatenation, formatting, case conversion, etc.)

#### Step 2: Plan the Sequence

Arrange operations in the correct order (e.g., trim before comparing, format after concatenating)

#### Step 3: Handle Edge Cases

Consider null strings, empty strings, and special characters

#### Step 4: Verify Output Format

Ensure the result matches expected format (spaces, punctuation, case)

---

### 2.2 Array Problem-Solving Framework

#### Step 1: Understand Array Structure

Identify array size, data type, and initial values

#### Step 2: Plan Loop Strategy

Determine if you need to iterate forward, backward, or use specific indices

#### Step 3: Use Proper Indexing

Remember zero-based indexing: first element is [0], last is [length-1]

#### Step 4: Initialize Accumulator Variables

Set up variables for sum, count, max, min, etc. before the loop

#### Step 5: Handle Boundary Conditions

Check for empty arrays, single-element arrays, and index out of bounds

---

### 2.3 ArrayList Problem-Solving Framework

#### Step 1: Import and Initialize

Import ArrayList class and create instance with proper generic type

#### Step 2: Choose Appropriate Methods

Select from add(), get(), set(), remove(), size() based on requirements

#### Step 3: Handle Type Conversions

Convert between primitives and wrapper classes (int ↔ Integer)



#### Step 4: Iterate Safely

Use proper loop bounds with size() method, not length property

#### Step 5: Test Dynamic Operations

Verify add/remove operations work correctly and size updates

---

### Quick Problem-Solving Checklist

- ✓ Read the problem carefully and identify key requirements
  - ✓ Determine which data structure is most appropriate (String, Array, ArrayList)
  - ✓ Write pseudocode or outline the solution steps
  - ✓ Code the solution, paying attention to syntax and method names
  - ✓ Test with sample inputs mentally before finalizing
  - ✓ Check for common errors (off-by-one, null values, type mismatches)
- 

## 3. Solved Examples

### Example 1: String Concatenation [EASY]

**Problem:** Create a greeting message that says "Hello Alice! You are 25 years old."

**Given:** name = "Alice", age = 25

```
public class StringExample {  
    public static void main(String[] args) {  
        String name = "Alice";  
        int age = 25;  
        // Method 1: Using + operator  
        String greeting = "Hello " + name + "! You are " + age + " years old.";  
        // Method 2: Using String.format()  
        String greeting2 = String.format("Hello %s! You are %d years old.", name, age);  
        System.out.println(greeting);  
    }  
}
```

**\*\*Output:\*\***

...

Hello Alice! You are 25 years old.

### Example 2: String Modification [EASY]



**Problem:** Convert " hello world " to "HELLO WORLD" (remove spaces and convert to uppercase)

```
public class StringModification {  
  
    public static void main(String[] args) {  
  
        String input = " hello world ";  
  
        // Step 1: Remove leading/trailing spaces  
  
        String trimmed = input.trim();  
  
        // Step 2: Convert to uppercase  
  
        String result = trimmed.toUpperCase();  
  
        System.out.println("Original: '" + input + "'");  
  
        System.out.println("Result: '" + result + "'");  
  
    }  
}
```

**\*\*Output:\*\***

Original: ' hello world '

Result: 'HELLO WORLD'

---

### Example 3: Array Sum & Average [MEDIUM]

**Problem:** Calculate sum and average of array elements: {10, 20, 30, 40, 50}

```
public class ArrayOperations {  
  
    public static void main(String[] args) {  
  
        int[] numbers = {10, 20, 30, 40, 50};  
  
        // Calculate sum  
  
        int sum = 0;  
  
        for (int i = 0; i < numbers.length; i++) {
```



```
        sum += numbers[i];
    }

    // Calculate average

    double average = (double) sum / numbers.length;

    System.out.println("Sum: " + sum);

    System.out.println("Average: " + average);

}

}
```

**\*\*Output:\*\***

Sum: 150

Average: 30.0

---

#### Example 4: ArrayList Operations [MEDIUM]

**Problem:** Create an ArrayList, add numbers 1-5, remove element at index 2, and display all elements

```
import java.util.ArrayList;

public class ArrayListExample {

    public static void main(String[] args) {

        // Create ArrayList

        ArrayList<Integer> numbers = new ArrayList<>();

        // Add elements 1-5

        for (int i = 1; i <= 5; i++) {

            numbers.add(i);

        }

        System.out.println("Original list: " + numbers);
    }
}
```



```
// Remove element at index 2 (value 3)

numbers.remove(2);

System.out.println("After removal: " + numbers);

System.out.println("Size: " + numbers.size());

}

}

..
```

**\*\*Output:\*\***

...

Original list: [1, 2, 3, 4, 5]

After removal: [1, 2, 4, 5]

Size: 4

---

### Example 5: String Array Processing [HARD]

**Problem:** Given an array of names, create a formatted string with all names in uppercase, separated by commas

**Input:** {"alice", "bob", "charlie"}

**Expected Output:** "ALICE, BOB, CHARLIE"

```
public class StringArrayProcessing {

    public static void main(String[] args) {

        String[] names = {"alice", "bob", "charlie"};

        // Build result string

        String result = "";

        for (int i = 0; i < names.length; i++) {

            // Convert to uppercase

            String upperName = names[i].toUpperCase();
```



```
// Add to result
result += upperName;

// Add comma separator (except for last element)
if (i < names.length - 1) {
    result += ", ";
}
}

System.out.println(result);
}
}
```

**\*\*Output:\*\***

...

ALICE, BOB, CHARLIE

#### Key Concepts:

- Array iteration with proper index management
- String method application (toUpperCase)
- Conditional logic for separator (avoiding trailing comma)
- String concatenation in loops

---

## 4. Practice Questions

### Set A: Foundation Level (EASY)

#### Question 1 [EASY]

**Problem:** Write a Java program to concatenate two strings "Hello" and "World" with a space between them.

#### Answer Key:

```
String result = "Hello" + " " + "World";
```



**Output:** "Hello World"

---

## Question 2 [EASY]

**Problem:** Convert the string "programming" to uppercase.

**Answer Key:**

```
String result = "programming".toUpperCase();
```

**Output:** "PROGRAMMING"

---

## Question 3 [EASY]

**Problem:** Create an integer array of size 5 and initialize it with values 10, 20, 30, 40, 50. Print the third element.

**Answer Key:**

```
int[] arr = {10, 20, 30, 40, 50};
```

```
System.out.println(arr[2]);
```

**Output:** 30 (Remember: zero-based indexing!)

---

## Question 4 [EASY]

**Problem:** Find the length of the string "Java Programming".

**Answer Key:**

```
int length = "Java Programming".length();
```

**Output:** 16

---

## Question 5 [EASY]

**Problem:** Create an ArrayList of integers and add three numbers: 100, 200, 300. Print the first element.

**Answer Key:**

```
ArrayList<Integer> nums = new ArrayList<>();
```



```
nums.add(100);  
  
nums.add(200);  
  
nums.add(300);  
  
System.out.println(nums.get(0));
```

**Output:** 100

---

#### Question 6 [EASY]

**Problem:** Get the first character of the string "Hello".

**Answer Key:**

```
char firstChar = "Hello".charAt(0);
```

**Output:** 'H'

---

### Set B: Intermediate Level (MEDIUM)

#### Question 7 [MEDIUM]

**Problem:** Use String.format() to create a string: "Product: Laptop, Price: \$999.99"

**Answer Key:**

```
String result = String.format("Product: %s, Price: $%.2f", "Laptop", 999.99);
```

**Output:** "Product: Laptop, Price: \$999.99"

---

#### Question 8 [MEDIUM]

**Problem:** Create an ArrayList of strings, add three city names ("Paris", "London", "Tokyo"), and print the size.

**Answer Key:**

```
ArrayList<String> cities = new ArrayList<>();
```

```
cities.add("Paris");
```

```
cities.add("London");
```



```
cities.add("Tokyo");
```

```
System.out.println("Size: " + cities.size());
```

**Output:** Size: 3

---

### Question 9 [MEDIUM]

**Problem:** Write a program to find the maximum element in an array {15, 7, 23, 9, 31}.

**Answer Key:**

```
int[] arr = {15, 7, 23, 9, 31};
```

```
int max = arr[0];
```

```
for (int i = 1; i < arr.length; i++) {
```

```
    if (arr[i] > max) {
```

```
        max = arr[i];
```

```
    }
```

```
}
```

```
System.out.println("Maximum: " + max);
```

**Output:** Maximum: 31

---

### Question 10 [MEDIUM]

**Problem:** Count how many even numbers are in the array {1, 2, 3, 4, 5, 6, 7, 8}.

**Answer Key:**

```
int[] arr = {1, 2, 3, 4, 5, 6, 7, 8};
```

```
int count = 0;
```

```
for (int num : arr) {
```

```
    if (num % 2 == 0) {
```

```
        count++;
```



```
}  
  
}  
  
System.out.println("Even count: " + count);
```

**Output:** Even count: 4

---

### Question 11 [MEDIUM]

**Problem:** Extract substring "Pro" from "Programming" (characters from index 0 to 3).

**Answer Key:**

```
String sub = "Programming".substring(0, 3);
```

**Output:** "Pro"

---

### Question 12 [MEDIUM]

**Problem:** Replace all occurrences of 'a' with 'o' in the string "banana".

**Answer Key:**

```
String result = "banana".replaceAll("a", "o");
```

**Output:** "bonono"

---

## Set C: Advanced Level (HARD)

### Question 13 [HARD]

**Problem:** Write a program to reverse the words in a sentence. Input: "Java is fun", Output: "fun is Java"

**Answer Key:**

```
String input = "Java is fun";
```

```
String[] words = input.split(" ");
```

```
String result = "";
```

```
for (int i = words.length - 1; i >= 0; i--) {
```



```
result += words[i];  
  
if (i > 0) result += " ";  
  
}  
  
System.out.println(result);
```

**Output:** "fun is Java"

---

#### Question 14 [HARD]

**Problem:** Create a program that merges two sorted arrays {1, 3, 5} and {2, 4, 6} into a single ArrayList and displays it.

**Answer Key:**

```
int[] arr1 = {1, 3, 5};  
  
int[] arr2 = {2, 4, 6};  
  
ArrayList<Integer> merged = new ArrayList<>();  
  
for (int num : arr1) merged.add(num);  
  
for (int num : arr2) merged.add(num);  
  
Collections.sort(merged);  
  
System.out.println(merged);
```

**Output:** [1, 2, 3, 4, 5, 6]

---

#### Question 15 [HARD]

**Problem:** Write a program to check if a string is a palindrome (reads same forwards and backwards). Test with "radar".

**Answer Key:**

```
String str = "radar";  
  
String reversed = "";  
  
for (int i = str.length() - 1; i >= 0; i--) {
```



```
reversed += str.charAt(i);  
  
}  
  
boolean isPalindrome = str.equals(reversed);  
  
System.out.println("Is palindrome: " + isPalindrome);
```

**Output:** Is palindrome: true

---

### Practice Tips

- ✓ Start with Easy questions to build confidence
  - ✓ Time yourself - Easy: 1 min, Medium: 1.5 min, Hard: 2 min
  - ✓ Write code on paper first to simulate exam conditions
  - ✓ Check your answers carefully for syntax errors
  - ✓ Review concepts if you struggle with a question type
-



# SQL Query Fundamentals & Data Retrieval

## Database Schema Context:

All questions in this chapter refer to the following three-table database for a retail store.

### Products Table:

- product\_id (Integer, Primary Key)
- product\_name (String)
- category (String)
- price (Decimal)

### Customers Table:

- customer\_id (Integer, Primary Key)
- first\_name (String)
- last\_name (String)
- city (String)

### Orders Table:

- order\_id (Integer, Primary Key)
- customer\_id (Foreign Key)
- order\_date (Date: YYYY-MM-DD)
- amount (Decimal)

## 1. Essential Concepts & Terminology

Master these ten fundamental SQL clauses and functions. They are the building blocks for all queries.

### SELECT

- **Definition:** Specifies the columns you want to retrieve from the database.
- **General Form:** SELECT column1, column2 FROM ...
- **Example:** SELECT product\_name, price FROM Products;

### FROM

- **Definition:** Specifies the table (or tables) from which to retrieve the data.
- **General Form:** SELECT ... FROM table\_name;
- **Example:** SELECT \* FROM Customers;

### WHERE

- **Definition:** Filters rows based on specific conditions. It is applied *before* any grouping.
- **General Form:** SELECT ... FROM ... WHERE condition;
- **Example:** SELECT \* FROM Products WHERE category = 'Electronics';

### ORDER BY

- **Definition:** Sorts the final result set in ascending (ASC) or descending (DESC) order.
- **General Form:** SELECT ... FROM ... ORDER BY column\_name DESC;
- **Example:** SELECT \* FROM Products ORDER BY price DESC;



### Aggregate Functions (COUNT, SUM, AVG, MIN, MAX)

- **Definition:** Perform a calculation on a set of rows and return a single value.
- **General Form:** SELECT COUNT(column\_name) FROM ...
- **Example:** SELECT SUM(amount) FROM Orders;

### GROUP BY

- **Definition:** Groups rows that have the same values in specified columns into summary rows. Used with aggregate functions.
- **General Form:** SELECT category, COUNT(\*) FROM Products GROUP BY category;
- **Example:** SELECT category, AVG(price) FROM Products GROUP BY category;

### HAVING

- **Definition:** Filters *groups* based on conditions. It is applied *after* grouping (unlike WHERE).
- **General Form:** SELECT ... GROUP BY ... HAVING condition;
- **Example:** SELECT category, COUNT(\*) FROM Products GROUP BY category HAVING COUNT(\*) > 10;

### INNER JOIN

- **Definition:** Combines rows from two tables, returning only those rows where the join condition is met in *both* tables.
- **General Form:** SELECT ... FROM table1 INNER JOIN table2 ON table1.column = table2.column;
- **Example:** SELECT \* FROM Orders INNER JOIN Customers ON Orders.customer\_id = Customers.customer\_id;

### LEFT JOIN

- **Definition:** Returns *all* rows from the left table, and the matched rows from the right table. If no match, NULL is returned for right table columns.
- **General Form:** SELECT ... FROM table1 LEFT JOIN table2 ON table1.column = table2.column;
- **Example:** SELECT \* FROM Customers LEFT JOIN Orders ON Customers.customer\_id = Orders.customer\_id;

### LIKE

- **Definition:** Used in a WHERE clause to search for a specified pattern in a column.
- **General Form:** SELECT ... WHERE column\_name LIKE 'pattern';
- **Example:** SELECT \* FROM Customers WHERE last\_name LIKE 'S%'; (finds last names starting with 'S')

## 1.1 Conceptual Framework & Problem-Solving Methodology

This section explains *how* SQL thinks, which is essential for writing complex queries correctly.

### 1.1 Fundamental Framework: Logical Query Processing Order

This is the *most important* concept to understand for debugging. SQL is *written* in one order, but the database *executes* it in a different logical order:

1. **FROM / JOIN:** Gets the tables and combines them.
2. **WHERE:** Filters individual rows *before* any grouping.
3. **GROUP BY:** Collapses filtered rows into groups.
4. **HAVING:** Filters the *groups* themselves.
5. **SELECT:** Gathers the final columns (and runs functions like SUM, COUNT).



6. **ORDER BY:** Sorts the final output.
7. **LIMIT / OFFSET:** Slices the sorted output.

**Key Insight:** This is why you **cannot** use an alias from your SELECT clause in your WHERE clause. The WHERE clause runs *before* the SELECT clause.

## 1.2 Methodology for Basic Retrieval (Filtering & Sorting)

**Objective:** To get a specific subset of raw data.

**Steps:**

- **Identify Columns:** What do you need to see? -> SELECT product\_name, price
- **Identify Table:** Where does this data live? -> FROM Products
- **Identify Row Filters:** What rows do you need? -> WHERE category = 'Electronics' AND price > 500
- **Identify Sorting:** How should it be ordered? -> ORDER BY price DESC

**Example Problem:** Find the names and prices of all products in the 'Electronics' category that cost more than \$500, sorted most expensive first.

**Solution:**

```
SELECT product_name, price
FROM Products
WHERE category = 'Electronics' AND price > 500
ORDER BY price DESC;
```

## 1.3 Methodology for Data Aggregation (Grouping)

- **Objective:** To calculate summaries (counts, sums, averages) for categories of data.
- **Key Distinction: WHERE vs. HAVING**
  - WHERE filters **rows** (e.g., WHERE category = 'Electronics'). It runs *before* GROUP BY.
  - HAVING filters **groups** (e.g., HAVING COUNT(\*) > 10). It runs *after* GROUP BY.
- **Steps:**
  0. **Identify Grouping Column:** What defines the category? -> category
  1. **Identify Aggregate Metric:** What do you need to calculate? -> COUNT(product\_id)
  2. **Build Query:** SELECT category, COUNT(product\_id) FROM Products GROUP BY category;
  3. **Add Group Filter (if needed):** "Show only categories with more than 10 products."
  4. **Final Query:**
  5. SELECT category, COUNT(product\_id) AS product\_count
  6. FROM Products
  7. GROUP BY category
  8. HAVING COUNT(product\_id) > 10;

## 1.4 Methodology for Combining Data (Joins)

**Objective:** To combine data from two or more tables based on a shared key (e.g., customer\_id).

**INNER JOIN (The Default)**

- **Use Case:** When you *only* want results where a match exists in **both** tables.
- **Example:** "List all orders and the names of the customers who placed them." If a customer hasn't ordered, they are excluded.
- **Methodology:**



1. Start with the "many" table (Orders).
2. INNER JOIN the "one" table (Customers).
3. Specify the ON condition (the shared key): Orders.customer\_id = Customers.customer\_id.

o **Solution:**

- o SELECT O.order\_id, C.first\_name, C.last\_name, O.amount
  - o FROM Orders AS O
  - o INNER JOIN Customers AS C ON O.customer\_id = C.customer\_id;
- (Note: O and C are table aliases to make the query shorter.)

### LEFT JOIN (Finding What's Missing)

- o **Use Case:** When you want *all* records from one table (the "left" one) and *only* the matching records from the other.
  - o **Example:** "List *all* customers, and show their order details *if* they have any."
  - o **Methodology:**
    1. Start with the table you want all records from (Customers).
    2. LEFT JOIN the table that might be missing data (Orders).
    3. The ON condition is the same.
  - o **Solution:**
  - o SELECT C.first\_name, C.last\_name, O.order\_id, O.amount
  - o FROM Customers AS C
  - o LEFT JOIN Orders AS O ON C.customer\_id = O.customer\_id;
- (A customer with no orders will appear, but order\_id and amount will be NULL.)

---

## 2. Solved Examples

**Difficulty Legend:** ★ Easy (1-2min) | ★★ Medium (3-4min) | ★★★ Hard (5-7min)

### Question 1: (★) Basic Selection & Logic

**Problem:** Retrieve the names and prices of all products in the 'Electronics' category that cost \$100 or less.

**Answer:**

- SELECT product\_name, price
- FROM Products
- WHERE category = 'Electronics' AND price <= 100;

**Method Analysis:**

- o SELECT product\_name, price: Specifies the output columns.
- o FROM Products: Specifies the source table.
- o WHERE ...: Filters rows *before* they are returned.
- o AND: Ensures *both* conditions (category and price) must be true.

---

### Question 2: (★) Sorting & Limiting

**Problem:** Identify the top 3 most expensive products in the store.

**Answer:**

- SELECT product\_name, price
- FROM Products



- ORDER BY price DESC
- LIMIT 3;

**Method Analysis:**

- ORDER BY price DESC: Sorts all products from highest to lowest price. This *must* happen before LIMIT.
  - LIMIT 3: Takes only the first 3 rows from the sorted list.
- 

**Question 3: (★★) Pattern Matching**

**Problem:** Find all customers whose last name starts with the letter 'S'.

**Answer:**

- SELECT first\_name, last\_name
- FROM Customers
- WHERE last\_name LIKE 'S%';

**Method Analysis:**

- LIKE: The standard operator for pattern matching.
- 'S%': The pattern. S means "must start with S". % is a wildcard for "zero or more characters."

**Question 4: (★★) Grouping and Counting**

**Problem:** Find the total number of products in each category.

**Answer:**

- SELECT category, COUNT(product\_id) AS product\_count
- FROM Products
- GROUP BY category;

**Method Analysis:**

- GROUP BY category: Collapses all 'Electronics' rows into one group, 'Clothing' into another, etc.
  - COUNT(product\_id): Counts the products *within* each group.
  - AS product\_count: An alias to make the output column name clean.
- 

**Question 5: (★★) Basic Aggregates (No Grouping)**

**Problem:** What is the total revenue (sum of amounts) and the average order value for the entire store?

**Answer:**

- SELECT SUM(amount) AS total\_revenue, AVG(amount) AS average\_order\_value
- FROM Orders;

**Method Analysis:**

- When aggregate functions are used *without* a GROUP BY clause, they run across the entire table and return a single row of results.
- 

**Question 6: (★★) Filtering with HAVING**

**Problem:** Show only the categories that have more than 10 products.

**Answer:**

- SELECT category, COUNT(product\_id) AS product\_count
- FROM Products



- GROUP BY category
- HAVING COUNT(product\_id) > 10;

**Method Analysis:**

- This query first groups all products by category and counts them (like Q4).
  - HAVING ...: This clause runs *after* grouping and filters out any groups where the count is not greater than 10.
- 

**Question 7: (★★★) INNER JOIN**

**Problem:** List all orders (order ID, date, amount) along with the full name of the customer who placed them.

**Answer:**

- SELECT
- O.order\_id,
- O.order\_date,
- O.amount,
- C.first\_name,
- C.last\_name
- FROM Orders AS O
- INNER JOIN Customers AS C ON O.customer\_id = C.customer\_id;

**Method Analysis:**

- INNER JOIN is used because we only care about orders that *have* a matching customer.
  - ON O.customer\_id = C.customer\_id is the "join key" that links the two tables.
- 

**Question 8: (★★★) LEFT JOIN (Finding Missing Data)**

**Problem:** Find all customers who have *never* placed an order.

**Answer:**

- SELECT C.first\_name, C.last\_name
- FROM Customers AS C
- LEFT JOIN Orders AS O ON C.customer\_id = O.customer\_id
- WHERE O.order\_id IS NULL;

**Method Analysis:**

- LEFT JOIN: Gets *all* customers (left table) and their matching orders (right table).
  - For customers with no orders, all columns from Orders (like O.order\_id) will be NULL.
  - WHERE O.order\_id IS NULL: This filters the joined result to *only* show those customers with NULL in the order column.
- 

**Question 9: (★★★) JOIN + GROUP BY**

**Problem:** Calculate the total money spent by each customer, listed by name.

**Answer:**

- SELECT
- C.first\_name,
- C.last\_name,
- SUM(O.amount) AS total\_spent
- FROM Customers AS C
- INNER JOIN Orders AS O ON C.customer\_id = O.customer\_id
- GROUP BY C.customer\_id, C.first\_name, C.last\_name;



### Method Analysis:

- INNER JOIN: Connects customers to their orders.
- GROUP BY C.customer\_id...: Collapses all orders for *each* customer into a single group.
- SUM(O.amount): Calculates the sum *within* each customer's group.
- **Rule:** Any non-aggregated column in the SELECT (like first\_name) *must* be in the GROUP BY.

## 3. Practice Questions & Self Assessment

Complete each set in the allocated time. Answers provided at the end.

### SET A: Foundation Level (Target: 5 minutes total)

1. List all customer first names, last names, and cities.
2. Find all orders with an amount over \$200.
3. List all products in the 'Clothing' category, sorted by price from cheapest to most expensive (ASC).
4. Find all customers who live in 'Chicago'.

### SET B: Intermediate Level (Target: 8 minutes total)

5. Find the single most expensive order (highest amount) in the Orders table.
6. How many customers are there in each city?
7. Find all products whose name *contains* the word 'Phone'. (Hint: LIKE can use wildcards on both sides).
8. What is the average price of products in the 'Electronics' category?

### SET C: Advanced Level (Target: 10 minutes total)

9. List all customers and their order dates. Include customers who have not placed any orders.
10. Find the total sales (sum of amount) for the month of November 2023. (Hint: Use WHERE order\_date >= '2023-11-01' AND order\_date <= '2023-11-30').
11. List the names of all customers who have spent more than \$500 in total (use the query from Q9 as a base).

## Answer KEY & BRIEF SOLUTIONS

Q #	Answer	Key Concept / Pattern
A1	SELECT first_name, last_name, city FROM Customers;	Basic SELECT
A2	SELECT * FROM Orders WHERE amount > 200;	WHERE with numeric filter
A3	SELECT * FROM Products WHERE category = 'Clothing' ORDER BY price ASC;	WHERE and ORDER BY ASC
A4	SELECT * FROM Customers WHERE city = 'Chicago';	WHERE with string filter
B5	SELECT * FROM Orders ORDER BY amount DESC LIMIT 1;	ORDER BY + LIMIT
B6	SELECT city, COUNT(customer_id) FROM Customers GROUP BY city;	GROUP BY with COUNT



B7	SELECT * FROM Products WHERE product_name LIKE '%Phone%';	LIKE with % wildcard
B8	SELECT AVG(price) FROM Products WHERE category = 'Electronics';	AVG aggregate with WHERE filter
C9	SELECT C.first_name, O.order_date FROM Customers C LEFT JOIN Orders O ON C.customer_id = O.customer_id;	LEFT JOIN
C10	SELECT SUM(amount) FROM Orders WHERE order_date BETWEEN '2023-11-01' AND '2023-11-30';	SUM with WHERE date range
C11	SELECT C.first_name, SUM(O.amount) FROM Customers C JOIN Orders O ON C.customer_id = O.customer_id GROUP BY C.customer_id, C.first_name HAVING SUM(O.amount) > 500;	JOIN + GROUP BY + HAVING



# Quantitative Reasoning

## 1. Essential Concepts & Terminology

### 1.1 Percentage

**Definition:** A percentage is a fraction or ratio expressed as a part of 100, denoted by the symbol "%".

$$\text{Percentage} = (\text{Part} / \text{Whole}) \times 100$$

**Key Properties:**

- Percentages are dimensionless quantities.
- Can be converted to decimals by dividing by 100.
- Used for comparing proportions and calculating changes.

**Example:** If 30 out of 150 students passed an exam, the pass percentage is  $(30/150) \times 100 = 20\%$

### 1.2 Ratio

**Definition:** A ratio is a comparison of two quantities by division, typically expressed as a:b.

Ratio = a:b where a and b are quantities being compared

**Key Properties:**

- Ratios can be simplified like fractions
- The order matters ( $a : b \neq b : a$  unless  $a = b$ )
- Can be expressed as fractions:  $a : b = a / b$

**Example:** If a mixture contains 2 liters of water and 3 liters of milk, the ratio of water to milk is 2 : 3



### 1.3 Proportion

**Definition:** A proportion is an equation stating that two ratios are equal.

$a:b = c:d$  or  $a/b = c/d$

**Key Properties:**

- Cross multiplication:  $a \times d = b \times c$
- Used to solve for unknown quantities
- Direct and inverse proportions

**Example:** If  $3:4 = 6:x$ , then  $3x = 24$ , so  $x = 8$

### 1.4 Average (Arithmetic Mean)

**Definition:** The average is the sum of all values divided by the number of values.

Average = (Sum of all values) / (Number of values)

**Key Properties:**

- Represents the central tendency of data
- Sensitive to extreme values (outliers)
- Can be used for weighted averages

**Example:** Average of 10, 20, 30, 40 is  $(10+20+30+40) / 4 = 25$

### 1.5 Simple Interest

**Definition:** Interest calculated only on the principal amount throughout the investment period.

$SI = (P \times R \times T) / 100$

Where P = Principal, R = Rate of interest per annum, T = Time in years

**Key Properties:**

- Interest remains constant each year
- Total Amount = Principal + Simple Interest
- Linear growth over time

**Example:** ₹1000 at 10% per annum for 2 years:  $SI = (1000 \times 10 \times 2)/100 = ₹200$

### 1.6 Compound Interest

**Definition:** Interest calculated on both principal and accumulated interest from previous periods.

$A = P(1 + r/n)^{nt}$  CI = A - P

Where A = Final amount, P = Principal, r = annual rate, n = compounding frequency, t = time in years

**Key Properties:**

- Interest grows exponentially
- Greater returns than simple interest over time
- Can compound annually, semi-annually, quarterly, etc.

**Example:** ₹1000 at 10% compounded annually for 2 years:  $A = 1000(1.1)^2 = ₹1210$ ,  $CI = ₹210$

## 1.7 Profit and Loss

**Definition:** The difference between cost price (CP) and selling price (SP) of goods.

Profit = SP - CP (when SP > CP) Loss = CP - SP (when CP > SP)

Profit% =  $(\text{Profit}/\text{CP}) \times 100$  Loss% =  $(\text{Loss}/\text{CP}) \times 100$

**Key Properties:**

- Always calculated on Cost Price unless specified
- Discount is given on Marked Price
- Can involve successive transactions

**Example:** CP = ₹500, SP = ₹600, Profit = ₹100, Profit% =  $(100/500) \times 100 = 20\%$

## 1.8 Speed, Distance, and Time

**Definition:** Relationship between the rate of motion, distance covered, and time taken.

Speed = Distance / Time Distance = Speed × Time Time = Distance / Speed

**Key Properties:**

- Units must be consistent (km/hr, m/s, etc.)
- Average speed = Total distance / Total time
- Relative speed: Same direction ( $S_1 - S_2$ ), Opposite direction ( $S_1 + S_2$ )

**Example:** A car travels 120 km in 2 hours. Speed =  $120/2 = 60$  km/hr

## 1.9 Permutations

**Definition:** The number of ways to arrange 'r' objects from 'n' objects where order matters.

$P(n, r) = n! / (n-r)!$

**Key Properties:**

- Order is important (ABC ≠ BAC)
- $P(n, n) = n!$  (all objects arranged)
- Used in ranking, seating arrangements



**Example:** Number of ways to arrange 3 books from 5 books =  $5!/(5-3)! = 5!/2! = 60$

## 1.10 Combinations

**Definition:** The number of ways to select 'r' objects from 'n' objects where order doesn't matter.

$$C(n,r) = n! / [r!(n-r)!]$$

**Key Properties:**

- Order is not important ( $ABC = BAC = CAB$ )
- $C(n,r) = C(n, n-r)$
- Used in selection, committee formation

**Example:** Selecting 3 students from 5 students =  $5!/(3! \times 2!) = 10$  ways

## 1.11 Probability

**Definition:** The likelihood of an event occurring, expressed as a number between 0 and 1.

$$P(E) = (\text{Number of favorable outcomes}) / (\text{Total number of outcomes})$$

**Key Properties:**

- $0 \leq P(E) \leq 1$
- $P(\text{certain event}) = 1$ ,  $P(\text{impossible event}) = 0$
- $P(\text{not } E) = 1 - P(E)$

For independent events:  $P(A \text{ and } B) = P(A) \times P(B)$

**Example:** Probability of getting a head when tossing a fair coin =  $1/2 = 0.5$

## 1.12 Arithmetic Progression (AP)

**Definition:** A sequence where the difference between consecutive terms is constant.

$$n\text{th term: } a_n = a + (n-1)d \quad \text{Sum of } n \text{ terms: } S_n = n/2[2a + (n-1)d] \text{ or } n/2[a + l]$$

Where  $a$  = first term,  $d$  = common difference,  $l$  = last term

**Key Properties:**

- Common difference  $d = a_{n+1} - a_n$
- Each term is average of its neighbors
- Used in linear patterns



**Example:** 2, 5, 8, 11, 14... ( $a=2$ ,  $d=3$ ), 10th term =  $2+(10-1) \times 3 = 29$

### 1.13 Geometric Progression (GP)

**Definition:** A sequence where the ratio between consecutive terms is constant.

$n$ th term:  $a_n = a \times r^{(n-1)}$  Sum of  $n$  terms:  $S_n = a(r^n - 1)/(r - 1)$  when  $r \neq 1$

Where  $a$  = first term,  $r$  = common ratio

**Key Properties:**

- Common ratio  $r = a_{n+1}/a_n$
- Each term is geometric mean of its neighbors
- Used in exponential growth/decay

**Example:** 3, 6, 12, 24, 48... ( $a=3$ ,  $r=2$ ), 6th term =  $3 \times 2^5 = 96$

### 1.14 Number System Properties

**Definition:** Fundamental properties and classifications of numbers.

**Key Classifications:**

- **Natural Numbers (N):** 1, 2, 3, 4, 5...
- **Whole Numbers (W):** 0, 1, 2, 3, 4...
- **Integers (Z):** ...-3, -2, -1, 0, 1, 2, 3...
- **Rational Numbers (Q):** Can be expressed as  $p/q$  where  $q \neq 0$
- **Irrational Numbers:** Cannot be expressed as  $p/q$  ( $\sqrt{2}$ ,  $\pi$ ,  $e$ )
- **Real Numbers (R):** All rational and irrational numbers
- **Prime Numbers:** Natural numbers  $> 1$  with exactly two factors (1 and itself)
- **Composite Numbers:** Natural numbers  $> 1$  that are not prime

**Divisibility Rules:**

- **Divisible by 2:** Last digit is even
- **Divisible by 3:** Sum of digits divisible by 3
- **Divisible by 5:** Last digit is 0 or 5
- **Divisible by 9:** Sum of digits divisible by 9
- **Divisible by 10:** Last digit is 0



## 2. Conceptual Framework & Problem-Solving Methodology

### 2.1 Basic Algebra & Arithmetic

#### 2.1.1 Linear Equations

**Concept:** Equations of the form  $ax + b = c$  where  $a$ ,  $b$ ,  $c$  are constants and  $x$  is the variable.

**Solution Methodology:**

1. Isolate the variable term on one side
2. Isolate constants on the other side
3. Divide both sides by the coefficient of the variable
4. Verify the solution by substituting back

**Example:** Solve  $3x + 5 = 20$

**Solution:**

$$3x + 5 = 20 \quad 3x = 20 - 5 \quad 3x = 15$$

$$x = 5$$

**Common Errors:**

- Sign errors when transposing terms
- Forgetting to perform the same operation on both sides
- Division by zero

#### 2.1.2 Quadratic Equations

**Concept:** Equations of the form  $ax^2 + bx + c = 0$  where  $a \neq 0$ .

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

**Solution Methodology:**

1. Identify coefficients  $a$ ,  $b$ , and  $c$
2. Calculate discriminant:  $\Delta = b^2 - 4ac$
3. If  $\Delta > 0$ : Two distinct real roots
4. If  $\Delta = 0$ : One repeated real root
5. If  $\Delta < 0$ : No real roots (complex roots)
6. Apply quadratic formula or factorization

#### 2.1.3 Simplification and BODMAS

**Order of Operations (BODMAS/PEMDAS):**



1. Brackets / Parentheses
2. Orders / Exponents (powers, roots)
3. Division and Multiplication (left to right)
4. Addition and Subtraction (left to right)

**Example:** Simplify:  $3 + 4 \times (2^2 - 1) \div 2$

**Solution:**

$$= 3 + 4 \times (4 - 1) \div 2 = 3 + 4 \times 3 \div 2$$

$$= 3 + 12 \div 2 = 3 + 6$$

$$= 9$$

## 2.2 Percentage, Ratio & Averages

### 2.2.1 Percentage Change Problems

**Formula for Percentage Change:**

$$\text{Percentage Change} = [(\text{New Value} - \text{Original Value}) / \text{Original Value}] \times 100$$

**Successive Percentage Changes:**

$$\text{If two changes are } a\% \text{ and } b\%, \text{ net change} = [a + b + (ab/100)]\%$$

**Example:** A price is increased by 20% and then decreased by 20%. What is the net change?

**Solution:**

$$\text{Net change} = [20 + (-20) + (20 \times (-20)/100)]\% = [0 - 400 / 100]\%$$

$$= -4\%$$

Therefore, there is a net decrease of 4%

### 2.2.2 Ratio and Proportion Applications

**Problem-Solving Steps:**

1. Identify the given ratio and total quantity
2. Convert ratio to parts (a:b means a+b total parts)
3. Calculate value of one part
4. Multiply by the required number of parts

**Example:** Divide ₹1200 in the ratio 2:3:5

**Solution:**

$$\text{Total parts} = 2 + 3 + 5 = 10 \quad \text{Value of 1 part} = 1200/10 = ₹120 \quad \text{First share} = 2 \times 120 =$$

$$₹240 \quad \text{Second share} = 3 \times 120 = ₹360 \quad \text{Third share} = 5 \times 120 = ₹600$$



### 2.2.3 Weighted Averages

**Formula:**

$$\text{Weighted Average} = (w_1 \times x_1 + w_2 \times x_2 + \dots + w_n \times x_n) / (w_1 + w_2 + \dots + w_n)$$

**Example:** A student scores 80 in Math (weight 3), 70 in Science (weight 2), and 90 in English (weight 1). Find weighted average.

**Solution:**

$$\begin{aligned} \text{Weighted Average} &= (3 \times 80 + 2 \times 70 + 1 \times 90) / (3 + 2 + 1) = (240 + 140 + 90) / 6 \\ &= 470 / 6 = 78.33 \end{aligned}$$

## 2.3 Number Systems & Patterns

### 2.3.1 Pattern Recognition

**Types of Number Patterns:**

- **Arithmetic Sequences:** Constant difference between terms
- **Geometric Sequences:** Constant ratio between terms
- **Square/Cube Sequences:** 1, 4, 9, 16... or 1, 8, 27, 64...
- **Fibonacci-type:** Each term is sum of previous two terms
- **Prime Number Patterns:** 2, 3, 5, 7, 11, 13...

**Example:** Find the next term: 2, 6, 12, 20, 30, ?

**Solution:**

Differences: 4, 6, 8, 10, ...

Second differences: 2, 2, 2 (constant) Next difference: 12

Next term:  $30 + 12 = 42$

### 2.3.2 Divisibility and Remainders

**Key Concepts:**

- **Division Algorithm:** Dividend = (Divisor  $\times$  Quotient) + Remainder
- **LCM (Least Common Multiple):** Smallest number divisible by all given numbers
- **HCF/GCD (Highest Common Factor):** Largest number that divides all given numbers

**Relationship:**

For two numbers a and b:  $\text{LCM}(a,b) \times \text{HCF}(a,b) = a \times b$



## 2.4 Probability & Counting

### 2.4.1 Basic Counting Principles

**Addition Principle:** If event A can occur in  $m$  ways and event B can occur in  $n$  ways (mutually exclusive), then A or B can occur in  $m + n$  ways.

**Multiplication Principle:** If event A can occur in  $m$  ways and for each, event B can occur in  $n$  ways, then A and B can occur in  $m \times n$  ways.

**Example:** A restaurant offers 3 appetizers, 5 main courses, and 2 desserts. How many different meal combinations are possible? **Solution:**

Using multiplication principle:  $3 \times 5 \times 2 = 30$  combinations

### 2.4.2 Probability of Combined Events

#### Independent Events

$$P(A \text{ and } B) = P(A) \times P(B)$$

#### Mutually Exclusive Events:

$$P(A \text{ or } B) = P(A) + P(B)$$

#### Complementary Events:

$$P(\text{not } A) = 1 - P(A)$$

## 2.5 Mathematical Reasoning

### 2.5.1 Logical Deduction

#### Types of Reasoning:

- **Deductive Reasoning:** Drawing specific conclusions from general principles
- **Inductive Reasoning:** Drawing general conclusions from specific observations
- **Analogical Reasoning:** Drawing conclusions based on similarities

### 2.5.2 Mathematical Statements

#### Types:

- **Theorem:** A statement that can be proven true
- **Axiom/Postulate:** A statement accepted as true without proof
- **Conjecture:** A statement believed to be true but not yet proven
- **Corollary:** A result that follows directly from a theorem

### 2.5.3 Problem-Solving Strategies

1. **Understand the Problem:** Identify what is given and what is asked
2. **Devise a Plan:** Select appropriate strategy or formula
3. **Execute the Plan:** Perform calculations systematically
4. **Verify the Solution:** Check if answer makes sense and satisfies condition



### 3. Solved Examples With Detailed Explanations

#### Example 1: Linear Equation [Easy - 1.5 minutes]

**Question:** If  $5x - 3 = 2x + 12$ , find the value of  $x$ .

**Solution:**

**Step 1:** Bring all terms with  $x$  to the left side

$$5x - 2x = 12 + 3$$

**Step 2:** Simplify

$$3x = 15$$

**Step 3:** Divide both sides by 3

$$x = 5$$

**Verification:**  $5(5) - 3 = 25 - 3 = 22$ ;  $2(5) + 12 = 10 + 12 = 22$  ✓

**Answer:**  $x = 5$

**Common Errors:** Sign mistakes when transposing terms

#### Example 2: Percentage [Easy - 2 minutes]

**Question:** A shirt originally priced at ₹800 is offered at a 25% discount. What is the sale price?

**Solution:**

**Method 1 (Direct):**

$$\text{Discount amount} = 25\% \text{ of } 800 = (25/100) \times 800 = ₹200$$

$$\text{Sale price} = 800 - 200 = ₹600$$

**Method 2 (Faster):**

If discount is 25%, customer pays 75%

$$\text{Sale price} = 75\% \text{ of } 800 = (75/100) \times 800 = ₹600$$

**Answer:** ₹600

**Time-Saving Tip:** Use Method 2 for faster calculation



### Example 3: Ratio & Proportion [Medium - 2.5 minutes]

**Question:** Three partners A, B, and C invest in a business in the ratio 2:3:5. If the total profit is ₹50,000, what is B's share?

**Solution:**

**Step 1:** Find total parts

$$\text{Total parts} = 2 + 3 + 5 = 10$$

**Step 2:** Calculate value of 1 part

$$1 \text{ part} = 50,000 \div 10 = ₹5,000$$

**Step 3:** Calculate B's share (3 parts)

$$\text{B's share} = 3 \times 5,000 = ₹15,000$$

**Answer:** ₹15,000

**Alternative Method:** B's share =  $(3/10) \times 50,000 = ₹15,000$

### Example 4: Average [Easy - 2 minutes]

**Question:** The average of 5 numbers is 40. If one number, 50, is removed, what is the new average?

**Solution:**

**Step 1:** Find sum of 5 numbers

$$\text{Sum} = \text{Average} \times \text{Count} = 40 \times 5 = 200$$

**Step 2:** Subtract removed number

$$\text{New sum} = 200 - 50 = 150$$

**Step 3:** Calculate new average

$$\text{New average} = 150 \div 4 = 37.5$$

**Answer:** 37.5

**Key Concept:** Sum = Average  $\times$  Number of values



### Example 5: Simple Interest [Medium - 2.5 minutes]

**Question:** At what rate of simple interest will ₹5,000 amount to ₹6,500 in 3 years?

**Solution:**

**Step 1:** Identify given values

Principal (P) = ₹5,000, Amount (A) = ₹6,500, Time (T) = 3 years

**Step 2:** Calculate Simple Interest

$$SI = A - P = 6,500 - 5,000 = ₹1,500$$

**Step 3:** Use SI formula to find rate

$$SI = (P \times R \times T) / 100$$

$$1,500 = (5,000 \times R \times 3) / 100$$

$$1,500 = 150R$$

$$R = 1,500 \div 150 = 10\%$$

**Answer:** 10% per annum

### Example 6: Probability [Medium - 2 minutes]

**Question:** A bag contains 3 red, 4 blue, and 5 green balls. If one ball is drawn at random, what is the probability that it is not green?

**Solution:**

**Step 1:** Find total number of balls

$$\text{Total balls} = 3 + 4 + 5 = 12$$

**Step 2:** Find number of balls that are not green

$$\text{Not green} = \text{Red} + \text{Blue} = 3 + 4 = 7$$

**Step 3:** Calculate probability

$$P(\text{not green}) = 7/12$$

**Alternative Method:**

$$P(\text{green}) = 5/12$$

$$P(\text{not green}) = 1 - P(\text{green}) = 1 - 5/12 = 7/12$$

**Answer:** 7/12



### Example 7: Number Pattern [Medium - 2.5 minutes]

**Question:** Find the missing number in the sequence: 3, 8, 15, 24, 35, ?

**Solution:**

**Step 1:** Analyze the pattern

$$3 = 1 \times 3 = (1)(1+2)$$

$$8 = 2 \times 4 = (2)(2+2)$$

$$15 = 3 \times 5 = (3)(3+2)$$

$$24 = 4 \times 6 = (4)(4+2)$$

$$35 = 5 \times 7 = (5)(5+2)$$

**Step 2:** Apply pattern for next term

$$\text{Next term} = 6 \times 8 = (6)(6+2) = 48$$

**Alternative Analysis:**

Differences: 5, 7, 9, 11, ... (arithmetic sequence with  $d=2$ )

Next difference = 13

$$\text{Next term} = 35 + 13 = 48$$

**Answer: 48**

### Example 8: Permutations & Combinations [Hard - 3 minutes]

**Question:** In how many ways can 4 students be selected from a group of 8 students for a debate team?

**Solution:**

**Step 1:** Identify the problem type

This is a combination problem (order doesn't matter)

**Step 2:** Apply combination formula

$$C(n,r) = n! / [r!(n-r)!]$$

$$C(8,4) = 8! / [4! \times 4!]$$

**Step 3:** Calculate

$$C(8,4) = (8 \times 7 \times 6 \times 5) / (4 \times 3 \times 2 \times 1)$$

$$= 1680 / 24$$

$$= 70$$

**Answer: 70 ways**

**Common Error:** Don't confuse with permutation (which would give 1680)



### Example 9: Speed, Distance, Time [Medium - 2.5 minutes]

**Question:** A train travels the first 120 km at 60 km/hr and the next 180 km at 90 km/hr. What is the average speed for the entire journey?

**Solution:**

**Step 1:** Calculate time for each segment

$$\text{Time}_1 = \text{Distance}_1 / \text{Speed}_1 = 120 / 60 = 2 \text{ hours}$$

$$\text{Time}_2 = \text{Distance}_2 / \text{Speed}_2 = 180 / 90 = 2 \text{ hours}$$

**Step 2:** Calculate total distance and time

$$\text{Total distance} = 120 + 180 = 300 \text{ km}$$

$$\text{Total time} = 2 + 2 = 4 \text{ hours}$$

**Step 3:** Calculate average speed

$$\text{Average speed} = \text{Total distance} / \text{Total time} = 300 / 4 = 75 \text{ km/hr}$$

**Answer:** 75 km/hr

**Important Note:** Average speed  $\neq$  Average of speeds (which would be 75 km/hr only by coincidence here)

### Example 10: Compound Interest [Hard - 3 minutes]

**Question:** Find the compound interest on ₹8,000 at 10% per annum for 2 years, compounded annually.

**Solution:**

**Step 1:** Identify values

$$P = ₹8,000, R = 10\% = 0.10, T = 2 \text{ years}, n = 1 \text{ (compounded annually)}$$

**Step 2:** Calculate final amount

$$A = P(1 + r)^t$$

$$A = 8,000(1 + 0.10)^2$$

$$A = 8,000(1.10)^2$$

$$A = 8,000 \times 1.21$$

$$A = ₹9,680$$

**Step 3:** Calculate compound interest

$$CI = A - P = 9,680 - 8,000 = ₹1,680$$

**Answer:** ₹1,680

**Comparison:** Simple Interest for same period =  $(8,000 \times 10 \times 2)/100 = ₹1,600$



# Data Structures & Algorithms

## 1. Algorithm Efficiency

### 1.1 Conceptual Notes

Algorithm efficiency refers to how well an algorithm uses computational resources such as time and memory. As input size ( $n$ ) grows, inefficient algorithms become impractical. **Big-O notation** provides an upper bound on time complexity, while **Big- $\Omega$**  gives a lower bound and **Big- $\Theta$**  represents a tight bound. Common complexity classes include  **$O(1)$**  (constant),  **$O(\log n)$**  (logarithmic),  **$O(n)$**  (linear),  **$O(n \log n)$** ,  **$O(n^2)$**  (quadratic), and exponential or factorial time algorithms like  **$O(2^n)$**  or  **$O(n!)$** .

Real-world analogies include choosing the fastest route in traffic or planning logistics where delays scale with distance or cargo size. For example, linear search scans every item in a list ( **$O(n)$** ), whereas binary search halves the search space each step ( **$O(\log n)$** ). Understanding complexity helps engineers design systems that scale under heavy load—like optimizing data pipelines or selecting appropriate sorting algorithms for large datasets.

### 1.2 Practice Questions and Solutions

#### 1. Nested loops complexity

**Question:** Consider the following nested-loop algorithm:

```
for i in range(n):  
    for j in range(n):  
        do constant work
```

What are the Big-O and Big- $\Theta$  time complexities?

**Solution:** The outer loop executes  $n$  iterations and for each iteration the inner loop also performs  $n$  iterations. The total number of constant-time operations is proportional to  $(n \text{ times } n = n^2)$ . Hence the time complexity is  **$O(n^2)$**  and  **$\Theta(n^2)$** .

#### 2. Comparing $O(n^2)$ vs $O(n \log n)$

**Question:** Algorithm A runs in  **$O(n^2)$**  time and Algorithm B runs in  **$O(n \log n)$**  time. For very large  $n$ , which algorithm is expected to be faster and why?

**Solution:** As  $n$  grows, the  $n \log n$  term grows much more slowly than  $n^2$ . Therefore Algorithm B ( **$O(n \log n)$** ) typically outperforms Algorithm A ( **$O(n^2)$** ) for large input sizes. For small  $n$ , constant factors may dominate, but asymptotically B is faster.



### 3. Maximum product pair

**Question:** Write a Python function to find the maximum product of two distinct numbers in an unsorted list and state its time complexity.

**Solution:** One approach is to sort the list and take the product of the two largest values. Sorting takes  $O(n \log n)$  time, and selecting the top two values is  $O(1)$ . A more efficient linear-time solution scans the list once to find the largest and second-largest values, achieving  $O(n)$  time and  $O(1)$  space.

### 4. Sum of array complexity

**Question:** Given the code:

```
total = 0
for value in data:
    total += value
```

What is the time complexity?

**Solution:** The loop iterates once per element in the list *data*, performing a constant-time addition each iteration. Thus the time complexity is  $O(n)$ . The space usage is  $O(1)$  because only a few variables are stored.

### 5. Quicksort complexities

**Question:** What are the average-case and worst-case time complexities of the quicksort algorithm?

**Solution:** Quicksort sorts an array by partitioning it into elements less than and greater than a pivot and recursively sorting each part. With a good pivot (e.g., median), the average-case complexity is  $O(n \log n)$ . In the worst case (e.g., if the smallest or largest element is always chosen as pivot), the complexity degrades to  $O(n^2)$ .

### 6. Merging k sorted lists

**Question:** Merging  $k$  sorted linked lists, each of length  $m$ , can be done by repeatedly selecting the smallest head element. What is the time complexity of this merging process using a min-heap?

**Solution:** Placing the initial heads into a min-heap of size  $k$  allows extraction of the smallest element in  $O(\log k)$  time. Each of the  $k$  lists has  $m$  elements, so there are  $k \cdot m$  total extractions and insertions. The overall time complexity is  $O(k \cdot m \log k)$ .

### 7. Merge sort space complexity

**Question:** What is the space complexity of merge sort?



**Solution:** Merge sort recursively divides an array and merges subarrays. It requires additional space for the temporary arrays used during merging. The space complexity is  $O(n)$  because it must allocate space proportional to the input size in the merge step.

## 8. Constant time example

**Question:** Give an example of an algorithm with constant ( $O(1)$ ) time complexity.

**Solution:** Accessing an element in an array by index, such as `arr[i]`, is  $O(1)$  because it requires a single memory lookup regardless of the array's size. Another example is swapping two variables.

## 9. Binary search complexity

**Question:** Binary search searches for a target in a sorted array by repeatedly halving the search space. Explain its time complexity.

**Solution:** Binary search compares the target to the middle element; if unequal, it discards half of the array and continues. Each comparison reduces the problem size by half, leading to a logarithmic time complexity of  $O(\log n)$ .

## 10. Hash set vs linked list lookup

**Question:** When checking membership of an element, why might a hash set be preferable to a linked list?

**Solution:** A hash set uses a hash function to achieve average-case  $O(1)$  membership tests. A linked list requires  $O(n)$  time to scan through elements. Thus a hash set offers dramatically faster lookups for large datasets.

## 11. Fibonacci algorithms

**Question:** Implement two methods to compute the  $n$ th Fibonacci number: one recursively and one iteratively. Compare their time complexities.

**Solution:** A naive recursive implementation that calls `fib(n-1) + fib(n-2)` exhibits exponential time complexity ( $O(\phi^n)$ ), where  $\phi$  is the golden ratio, due to repeated subproblem calculations. An iterative version or a version using dynamic programming (memoization) computes each value once, leading to  $O(n)$  time complexity and  $O(1)$  or  $O(n)$  space.

## 12. Feasibility of factorial-time algorithm

**Question:** How feasible is an algorithm with factorial time complexity ( $O(n!)$ ) when  $n = 10$ ?

**Solution:** An  $O(n!)$  algorithm generates all permutations of  $n$  items. When  $n = 10$ , there are ( $10! = 3,628,800$ ) permutations. This may be feasible on modern hardware, but time grows explosively: for  $n = 15$  there are over a trillion permutations, making the algorithm impractical. Such algorithms are typically used only for very small  $n$ .



### 13. Prefix sum algorithm

**Question:** Write an algorithm to compute the prefix sum (running sum) of a list and analyze its time complexity.

**Solution:** A simple loop that cumulatively sums the values can compute the prefix sums: initialize an empty result list and `running_sum = 0`; for each element `x`, set `running_sum += x` and append `running_sum` to the result list. This executes one constant-time operation per element, giving  **$O(n)$**  time and  **$O(n)$**  space complexity.

### 14. Master Theorem recurrence

**Question:** Solve the recurrence  $T(n) = 2T(n/2) + n$  using the Master Theorem.

**Solution:** In the Master Theorem form  $T(n) = aT(n/b) + f(n)$ , here  $(a=2)$ ,  $(b=2)$  and  $(f(n)=n)$ . Since  $(f(n) = \Theta(n^{\log_b a}))$  where  $(\log_b a = 1)$ , the recurrence falls into Case 2 of the Master Theorem, giving  **$T(n) = \Theta(n \log n)$** .

### 15. Higher Big-O may outperform lower Big-O

**Question:** Describe a scenario where an algorithm with a higher Big-O complexity might outperform one with lower complexity.

**Solution:** For small input sizes, constant factors and cache performance can outweigh theoretical growth rates. For example, insertion sort ( **$O(n^2)$** ) can outperform merge sort ( **$O(n \log n)$** ) on small arrays because it has minimal overhead and benefits from cache locality. Thus an algorithm with worse asymptotic complexity might be faster for small  $(n)$ .

---

### 1.3 Unsolved Practice Questions

- Determine the time and space complexity of an algorithm that scans an  $(n \text{ imes } m)$  matrix to find the maximum element.
- Compare the growth rates of functions  $(n \log n)$  and  $(n^{3/2})$ ; for which values of  $(n)$  does one dominate?
- Prove that the sum of the first  $n$  integers is  $\Theta(n^2)$  by using Big-O and Big- $\Omega$  definitions.
- Design an algorithm with  **$O(n)$**  time to find a missing number from an array containing values 1 through  $n$  with one missing.
- Analyze the time complexity of selection sort and explain its behaviour on already sorted input.
- Explain why an algorithm with time complexity  **$O(n \log n)$**  may have a larger constant factor than one with  **$O(n^2)$**  and how this affects runtime for small  $n$ .
- Given the recurrence  $T(n) = 3T(n/3) + n/n$ , use the Master Theorem to find its asymptotic complexity.
- What is the time complexity of computing the transpose of an  $(n \text{ imes } n)$  matrix? Justify your answer.
- Evaluate the space complexity of a recursive algorithm that calculates  $n!$  without returning intermediate values.



- Show that  $O(n) + O(\log n) = O(n)$ . Explain why lower-order terms are omitted in Big-O notation.
- 

## 2. Data Structures Fundamentals

### 2.1 Conceptual Notes

Data structures organize data to enable efficient access and modification. **Arrays** store elements in contiguous memory, supporting constant-time random access. **Linked lists** chain nodes together via pointers, allowing  $O(1)$  insertion or removal at known positions but  $O(n)$  indexing. **Stacks** (LIFO) and **queues** (FIFO) provide restricted access patterns analogous to stacks of plates or waiting lines.

**Hash tables** map keys to values via a hash function, achieving average-case  $O(1)$  lookup. **Trees** (e.g., binary search trees, heaps) impose hierarchical relationships; a BST allows  $O(\log n)$  search in the average case, while a heap supports efficient priority queue operations. **Graphs** model networks of nodes and edges, and can be represented via adjacency lists or matrices.

Choosing the right data structure is like choosing the right container in a kitchen: a bowl, jar or drawer — each is suited to different tasks. In software engineering, efficient data structures underlie caches, database indexes, and network routing tables.

### 2.2 Practice Questions and Solutions

#### 1. Stack via queues

**Question:** Implement a stack using two queues. Outline the operations push and pop and their complexities.

**Solution:** Use two queues, q1 and q2. For push(x), enqueue x into q2 and then dequeue all elements from q1 into q2; finally swap q1 and q2. For pop(), dequeue from q1. push is  $O(n)$  (because existing elements are moved), and pop is  $O(1)$ .

#### 2. Infix to postfix

**Question:** Convert an infix expression (e.g.,  $a + b * c$ ) to postfix using a stack.

**Solution:** Scan the expression. Output operands directly. For operators, pop from the stack to output while the top has greater or equal precedence, then push the new operator. At the end, pop remaining operators. Using a stack ensures proper order; the algorithm runs in  $O(n)$  time and uses  $O(n)$  space.

#### 3. Reverse a linked list

**Question:** Write code to reverse a singly linked list iteratively.



**Solution:** Initialize  $prev = None$  and  $current = head$ . While  $current$  is not  $None$ , store  $next\_node = current.next$ , set  $current.next = prev$ , update  $prev = current$  and  $current = next\_node$ . When finished,  $prev$  is the new head. The reversal runs in  $O(n)$  time and  $O(1)$  space.

#### 4. Cycle detection

**Question:** Describe how to detect a cycle in a linked list.

**Solution:** Use two pointers (Floyd's algorithm): a slow pointer moves one step and a fast pointer moves two steps. If the list has a cycle, the fast pointer will eventually meet the slow pointer. If fast reaches the end, the list has no cycle. This runs in  $O(n)$  time and  $O(1)$  space.

#### 5. kth smallest in BST

**Question:** Given a binary search tree (BST), write a function to find the  $k$ th smallest element.

**Solution:** Perform an in-order traversal, which visits nodes in sorted order. Maintain a counter and return the node when the counter reaches  $k$ . The traversal runs in  $O(h + k)$  time, where  $h$  is tree height, and  $O(h)$  space due to the call stack.

#### 6. LRU cache design

**Question:** Design an LRU (Least Recently Used) cache that supports get and put in  $O(1)$  time.

**Solution:** Use a hash map to store key  $\rightarrow$  node mappings and a doubly linked list to track usage order. On get/put, move the accessed node to the front (most recently used). When capacity is exceeded, remove the tail (least recently used). Both operations run in  $O(1)$  time.

#### 7. BFS with queue

**Question:** Use a queue to perform Breadth-First Search (BFS) on a graph. What is its time complexity?

**Solution:** Initialize a queue with the starting node and a visited set. Dequeue a node, process it, and enqueue its unvisited neighbors. Continue until the queue is empty. BFS runs in  $O(V + E)$  time where  $V$  is the number of vertices and  $E$  is the number of edges; it uses  $O(V)$  space for the queue and visited set.

#### 8. Min-heap operations

**Question:** Explain how to implement a min-heap and its key operations (insert and extract-min).

**Solution:** A min-heap is a complete binary tree stored in an array where each parent is  $\leq$  its children. To insert an element, append it to the array and "bubble up" while it is smaller than its parent. extract-min removes the root (smallest element), moves the last element to the root, and "heapifies" down until the heap property holds. Each operation runs in  $O(\log n)$  time.



## 9. Word frequency with hash table

**Question:** Use a hash table to count the frequency of each word in a large text.

**Solution:** Initialize an empty dictionary. For each word in the text, convert to a canonical form (e.g., lowercase) and increment `dict[word]` by one. This runs in  $O(n)$  time where  $n$  is the number of words, assuming average-case  $O(1)$  dictionary operations.

## 10. Sliding window maximum sum

**Question:** Given an array of integers, use the sliding window technique to find the contiguous subarray of size  $k$  with the maximum sum.

**Solution:** Compute the sum of the first  $k$  elements. Then slide the window: subtract the element leaving the window and add the new element entering. Track the maximum sum and corresponding indices. This runs in  $O(n)$  time and  $O(1)$  extra space.

## 11. Balanced parentheses

**Question:** Show how to check balanced parentheses in a string using a stack.

**Solution:** Scan characters and push opening brackets onto the stack. When encountering a closing bracket, pop from the stack and check for a matching type. If the stack is empty when a closing bracket appears or is not empty at the end, the string is unbalanced. This runs in  $O(n)$  time and uses  $O(n)$  space in the worst case.

## 12. Depth-first traversal

**Question:** Implement depth-first traversal of a binary tree both recursively and iteratively.

**Solution:** *Recursive:* define a helper function that visits the current node, then recurses on the left and right children. *Iterative:* use a stack; push the root, then repeatedly pop a node, process it, and push its right then left child. Both methods run in  $O(n)$  time and  $O(h)$  space, where  $h$  is the tree height (for recursion) or  $O(n)$  space for the stack in the worst case of a skewed tree.

## 13. Queue via stacks

**Question:** Construct a queue from two stacks and discuss the complexity of enqueue and dequeue.

**Solution:** Use two stacks  $s1$  and  $s2$ . For `enqueue(x)`, push  $x$  onto  $s1$ . For `dequeue`, if  $s2$  is empty, pop all elements from  $s1$  to  $s2$ . Then pop from  $s2$ . Amortized analysis shows both operations run in  $O(1)$  average time; in the worst case dequeue may take  $O(n)$  when transferring elements.

## 14. Insert into doubly linked list

**Question:** How do you insert a new element into a sorted doubly linked list?



**Solution:** Traverse the list to find the insertion point. Create a new node, set its prev pointer to the predecessor and next pointer to the successor, then adjust the neighboring nodes' pointers. This insertion runs in  $O(n)$  time for traversal and  $O(1)$  for pointer updates.

## 15. Graph representation

**Question:** Explain how to represent a weighted graph using adjacency lists and discuss the trade-offs compared to an adjacency matrix.

**Solution:** An adjacency list stores for each vertex a list of (neighbor, weight) pairs. It requires  $O(V + E)$  space and efficiently enumerates neighbors. An adjacency matrix uses  $O(V^2)$  space regardless of edges and allows  $O(1)$  edge existence checks but wastes space for sparse graphs. Adjacency lists are preferred for large sparse graphs.

## 2.3 Unsolved Practice Questions

- Implement a queue using a circular array and explain how to detect full vs empty states.
- Given two sorted linked lists, merge them into a single sorted list in  $O(n)$  time.
- Design a data structure that supports min, max, insert and delete in logarithmic time.
- Write a function to check whether a binary tree is height-balanced.
- Explain how a hash collision occurs and compare separate chaining vs open addressing.
- Implement a trie data structure for storing strings and support prefix search.
- Describe an algorithm to find all nodes at distance  $k$  from a given node in a binary tree.
- Given an undirected graph, determine whether it contains a cycle using union-find.
- Explain how a disjoint-set (union-find) data structure works and its applications.
- Create a dynamic array that resizes automatically and analyze the amortized cost of append operations.

# 3. Algorithmic Thinking

## 3.1 Conceptual Notes

Algorithmic thinking is the disciplined process of devising a step-by-step procedure to solve problems. It involves understanding the problem, exploring examples, identifying patterns, decomposing tasks, choosing appropriate data structures, selecting an algorithmic paradigm (such as greedy, divide-and-conquer, or dynamic programming), and rigorously analyzing and testing the solution.

Real-world analogies include planning a road trip by selecting routes, anticipating traffic, and optimizing travel time. By recognizing subproblems and constraints, engineers can design efficient solutions to tasks like resource scheduling, optimization, and data analysis. Algorithmic thinking promotes clarity, reusability, and scalability in software systems.



## 3.2 Practice Questions and Solutions

### 1. Task scheduling with deadlines

**Question:** Design a greedy algorithm to schedule tasks with deadlines and durations in order to maximize the number of tasks completed.

**Solution:** Sort tasks by their deadlines. Maintain a current time counter and a max-heap of durations. Iterate through tasks; for each task, add its duration to the heap and increase current time. If current time exceeds the task's deadline, remove the task with the longest duration from the heap. The size of the heap gives the maximum number of tasks. Complexity: sorting takes  $O(n \log n)$ ; heap operations take  $O(n \log n)$ .

### 2. Coin change with greedy

**Question:** Given coin denominations ([1, 5, 10, 25]), design an algorithm to make change for amount  $A$  using the fewest coins.

**Solution:** Use a greedy strategy: always take the largest coin less than or equal to the remaining amount until the amount is zero. This works for U.S. coin denominations because they are canonical. For arbitrary denominations, dynamic programming is needed. The greedy algorithm runs in  $O(n)$  relative to the number of coins used.

### 3. Anagram test

**Question:** Describe an algorithm to determine whether two strings are anagrams of each other.

**Solution:** Normalize both strings (e.g., lowercase, remove spaces). Count character frequencies using a hash map or sort both strings and compare. Counting runs in  $O(n)$  time and  $O(\sigma)$  space, where  $\sigma$  is the alphabet size. Sorting runs in  $O(n \log n)$  time.

### 4. Maximum subarray (Kadane)

**Question:** Solve the maximum subarray problem (Kadane's algorithm).

**Solution:** Initialize `max_so_far` and `current_sum` to the first element. Iterate through the array: at each element  $x$ , set `current_sum` =  $\max(x, \text{current\_sum} + x)$ , and update `max_so_far` =  $\max(\text{max\_so\_far}, \text{current\_sum})$ . The algorithm runs in  $O(n)$  time and  $O(1)$  space.

### 5. Product of array except self

**Question:** Find the product of all elements of an array except the current index without using division.

**Solution:** Compute prefix products: for each index  $i$ , store the product of elements before  $i$ . Compute suffix products: the product of elements after  $i$ . The result at  $i$  is `prefix[i] × suffix[i]`. This runs in  $O(n)$  time and uses  $O(n)$  extra space; an  $O(1)$  extra space variant uses two passes.



## 6. Two-sum problem

**Question:** Given an array of integers and a target sum  $k$ , devise an algorithm to determine whether any two numbers sum to  $k$ .

**Solution:** Insert each number into a hash set while iterating. For each number  $x$ , check if  $(k - x)$  exists in the set. If so, return True. This runs in  $O(n)$  time and  $O(n)$  space. Sorting the array and using a two-pointer technique also yields an  $O(n \log n)$  solution.

## 7. Longest common subsequence

**Question:** Outline the dynamic programming algorithm to compute the longest common subsequence (LCS) of two strings.

**Solution:** Create a 2D table  $dp$  of size  $((m+1) \times (n+1))$  where  $dp[i][j]$  is the length of the LCS of the first  $i$  characters of string  $A$  and the first  $j$  of  $B$ . Recurrence: if  $A[i-1] == B[j-1]$ , then  $dp[i][j] = dp[i-1][j-1] + 1$ ; else  $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$ . The algorithm runs in  $O(mn)$  time and  $O(mn)$  space; reconstructing the subsequence requires backtracking from  $dp[m][n]$ .

## 8. Path existence in graph

**Question:** Use BFS or DFS to determine whether there is a path between two nodes in an undirected graph.

**Solution:** Perform a breadth-first search starting from the source; if the destination is reached, return True; otherwise continue until the queue is empty. Alternatively, perform a depth-first search with recursion or a stack. Both methods run in  $O(V + E)$  time and  $O(V)$  space.

## 9. Activity selection

**Question:** Provide a greedy algorithm to select the maximum number of non-overlapping intervals.

**Solution:** Sort intervals by their finish times. Initialize  $count = 0$  and  $last\_finish = -\infty$ . Iterate through intervals; if an interval's start  $\geq last\_finish$ , select it and update  $last\_finish$  to its finish. This runs in  $O(n \log n)$  time due to sorting.

## 10. Generating permutations

**Question:** Implement a backtracking algorithm to generate all permutations of  $n$  distinct numbers.

**Solution:** Use recursion with a current path and a boolean array to mark used numbers. For each unused number, append it to the path, mark it used, recurse, and then backtrack. When the path length equals  $n$ , output the permutation. The algorithm explores  $n!$  permutations and uses  $O(n)$  space for the path.

## 11. Median of two sorted arrays

**Question:** Devise an algorithm to find the median of two sorted arrays of equal length.



**Solution:** Use a binary search approach: compare the medians of the two arrays and discard half of one array and half of the other accordingly. Recursively reduce the problem size until base cases. The algorithm runs in  $O(\log n)$  time. A more general solution uses median-finding in  $O(\log (m+n))$  time.

## 12. kth largest element

**Question:** Find the  $k$ th largest element in an unsorted array.

**Solution:** Use a min-heap of size  $k$ : traverse the array, pushing elements and popping when the heap size exceeds  $k$ . The root of the heap is the  $k$ th largest. This runs in  $O(n \log k)$  time. Alternatively, use Quickselect (a partition-based algorithm) which runs in average  $O(n)$  time.

## 13. Coin change ways

**Question:** Describe a dynamic programming solution to count the number of ways to make change for amount  $n$  with given coin denominations.

**Solution:** Let  $dp[i]$  be the number of ways to make amount  $i$ . Initialize  $dp[0] = 1$ . For each coin  $c$ , for  $i$  from  $c$  to  $n$ :  $dp[i] += dp[i - c]$ . This iterates through coins and amounts, running in  $O(n \times \text{number\_of\_coins})$  time.

## 14. Binary search for square root

**Question:** Use binary search to approximate the square root of a positive number  $x$ .

**Solution:** Define  $low = 0$  and  $high = \max(1, x)$ . While  $high - low > \epsilon$ , compute  $mid = (low + high) / 2$ . If  $mid^2 < x$ , set  $low = mid$ ; otherwise set  $high = mid$ . Return  $mid$  as the approximate square root. This runs in  $O(\log((b - a)/\epsilon))$  time.

## 15. Partition equal subset sum

**Question:** Given an array of integers, determine whether it can be partitioned into two subsets with equal sum.

**Solution:** Compute the total sum; if it is odd, partitioning is impossible. Otherwise, reduce to a subset sum problem: is there a subset with sum  $total/2$ ? Use dynamic programming:  $dp[i][j]$  indicates whether a sum  $j$  is achievable using the first  $i$  numbers. This runs in  $O(n \times total/2)$  time and space.

## 3.3 Unsolved Practice Questions

- Design a greedy algorithm for minimizing the number of platforms needed for trains given arrival and departure times.
- Given tasks with profits and deadlines, select tasks to maximize total profit (job sequencing with deadlines).
- Implement Dijkstra's algorithm to find shortest paths in a weighted graph.
- Devise a divide-and-conquer algorithm to count inversions in an array.
- Find the longest palindromic substring of a given string.
- Given a 2D grid of 0s and 1s, find the largest connected region of 1s using BFS or DFS.



- Solve the knapsack problem using dynamic programming.
- Implement the A\* search algorithm for pathfinding in a weighted grid.
- Determine the minimum number of edits (insertions, deletions, substitutions) to convert one string into another (edit distance).
- Write an algorithm to generate Gray codes for  $n$  bits and analyze its complexity.

## 4. Problem Decomposition

### 4.1 Conceptual Notes

Problem decomposition involves breaking a complex problem into simpler, more manageable subproblems. This facilitates understanding, reuse, parallelization and testing. Divide-and-conquer algorithms (like merge sort) split a problem, solve subproblems, and combine results. Dynamic programming solves overlapping subproblems by storing intermediate results. Recursion naturally expresses a problem in terms of smaller instances. In real life, assembling a car involves separate teams building the engine, chassis and electronics. In software, large systems are built from modules that each solve a subtask. Effective decomposition leads to cleaner code and scalable solutions.

### 4.2 Practice Questions and Solutions

#### 1. MapReduce word count

**Question:** Design a MapReduce-style decomposition to count the frequency of each word in a large document.

**Solution:** Split the document into chunks (*map* phase). Each mapper emits (word, 1) pairs. In the *shuffle* phase, group all pairs by word. The *reduce* phase sums the counts for each word. This decomposition allows parallel processing across multiple machines and runs in  $O(n/p)$  time per mapper for  $p$  processors, with an additional shuffle and reduce cost.

#### 2. Merge sort decomposition

**Question:** Describe how merge sort decomposes a sorting problem.

**Solution:** Merge sort recursively splits the array into two halves until subarrays of length  $\leq 1$  remain. It then merges sorted subarrays back together. The decomposition is captured by the recurrence  $T(n) = 2T(n/2) + n$ , yielding  $O(n \log n)$  time. Each level of recursion deals with smaller subproblems.

#### 3. Factorial: recursion vs iteration

**Question:** Explain how to compute  $n!$  using recursion versus iteration.



**Solution:** Recursive definition:  $n! = n \times (n-1)!$  with base case  $0! = 1$ . A recursive function returns  $n$  multiplied by factorial( $n-1$ ). An iterative approach multiplies numbers from 1 to  $n$  in a loop. Both produce the same result; the recursive method uses  **$O(n)$**  call stack space while the iterative version uses  **$O(1)$**  space.

#### 4. Quicksort decomposition

**Question:** How does quicksort use problem decomposition?

**Solution:** Quicksort selects a pivot and partitions the array into elements less than and greater than the pivot. It recursively sorts the subarrays. The base case is arrays of size 0 or 1. Decomposition yields the recurrence  **$T(n) = T(k) + T(n-k-1) + O(n)$** .

#### 5. Matrix chain multiplication

**Question:** Use dynamic programming to solve the matrix chain multiplication problem.

**Solution:** Let matrices  $A_1 \dots A_n$  have dimensions  $p_0 \times p_1, p_1 \times p_2, \dots, p_{n-1} \times p_n$ . Define  $dp[i][j]$  as the minimum multiplications needed to multiply  $A_i \dots A_j$ . Recurrence:  $dp[i][i] = 0$ ; for  $i < j$ ,  $dp[i][j] = \min_{k \text{ between } i \text{ and } j-1} \{ dp[i][k] + dp[k+1][j] + p_{i-1} \times p_k \times p_j \}$ . The solution takes  **$O(n^3)$**  time.

#### 6. Fibonacci with memoization

**Question:** Show how computing Fibonacci numbers can be decomposed and optimized with memoization.

**Solution:** The naive recursive definition  $fib(n) = fib(n-1) + fib(n-2)$  leads to overlapping subproblems. Memoization stores previously computed values in a dictionary. When  $fib(k)$  is called, return the stored result if present; otherwise compute and store it. This reduces the time complexity from exponential to  **$O(n)$** .

#### 7. Majority element via divide and conquer

**Question:** Use divide-and-conquer to find the majority element in an array.

**Solution:** Split the array into two halves. Recursively find the majority element in each half. If both halves agree, that element is the majority. Otherwise, count each candidate in the entire array and return the element with count  $> n/2$  if it exists; otherwise none. This runs in  **$O(n \log n)$**  time.

#### 8. Build BST from sorted array

**Question:** Describe how to build a binary search tree (BST) from a sorted array using decomposition.

**Solution:** Select the middle element as the root; recursively build the left subtree from the left half and the right subtree from the right half. This produces a balanced BST with height  **$O(\log n)$** .

#### 9. Tower of Hanoi



**Question:** Formulate the Tower of Hanoi problem as a recursive decomposition.

**Solution:** To move  $n$  disks from peg A to peg C using peg B: recursively move  $n-1$  disks from A to B, move the largest disk from A to C, then move the  $n-1$  disks from B to C. The recurrence  $T(n) = 2T(n-1) + 1$  yields  $T(n) = 2^n - 1$  moves.

## 10. BFS decomposition

**Question:** Decompose the BFS algorithm into its core operations.

**Solution:** BFS consists of: (1) enqueue the starting node; (2) while the queue is not empty: (a) dequeue a node, (b) process it, and (c) enqueue each unvisited neighbor. The decomposition highlights initialization, iteration and neighbor exploration.

## 11. Euclid's algorithm

**Question:** Show how Euclid's algorithm for GCD can be implemented recursively.

**Solution:** The greatest common divisor of  $a$  and  $b$  (with  $b \neq 0$ ) satisfies  $\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$ . Base case:  $\text{gcd}(a, 0) = a$ . Recursively apply the definition until  $b = 0$ . The algorithm uses  $O(\log \min(a, b))$  time.

## 12. Karatsuba multiplication

**Question:** Outline the Karatsuba algorithm for multiplying two large numbers.

**Solution:** Karatsuba's algorithm decomposes numbers into halves. Given  $x = x_1 \cdot 10^m + x_0$  and  $y = y_1 \cdot 10^m + y_0$ , compute  $z_2 = x_1 \cdot y_1$ ,  $z_0 = x_0 \cdot y_0$ , and  $z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$ . Then  $x \cdot y = z_2 \cdot 10^{2m} + z_1 \cdot 10^m + z_0$ . This reduces multiplication from four subproblems to three, giving  $O(n^{\log_2 3}) \approx O(n^{1.585})$ .

## 13. Largest rectangle in histogram

**Question:** Design a stack-based algorithm to find the largest rectangle under a histogram and describe its decomposition.

**Solution:** Maintain a stack of indices. Iterate through bars; while the current bar is shorter than the bar at the stack's top, pop the top and compute the area with that bar as height. Push the current index onto the stack. After processing all bars, pop remaining indices and compute areas. Decomposition involves handling each bar once; the algorithm runs in  $O(n)$  time.

## 14. Longest increasing subsequence

**Question:** Use dynamic programming to find the longest increasing subsequence (LIS).

**Solution:** Let  $\text{dp}[i]$  be the length of the LIS ending at index  $i$ . Initialize all  $\text{dp}[i] = 1$ . For each  $i$  from 1 to  $n-1$ , for each  $j < i$ : if  $\text{arr}[j] < \text{arr}[i]$ , update  $\text{dp}[i] = \max(\text{dp}[i], \text{dp}[j] + 1)$ . The answer is  $\max(\text{dp})$ . This runs in  $O(n^2)$  time. A more efficient  $O(n \log n)$  method uses patience sorting and binary search.

## 15. Convex hull via divide and conquer



**Question:** Explain the divide-and-conquer approach to computing the convex hull of a set of points.

**Solution:** Sort points by x-coordinate; split into left and right halves. Recursively compute the convex hull of each half. Merge the two hulls by finding the upper and lower tangents. The overall complexity is  $O(n \log n)$ .

### 4.3 Unsolved Practice Questions

- Develop a recursive algorithm to generate all balanced parentheses sequences of length  $2n$ .
  - Design a divide-and-conquer algorithm for the closest pair of points in a plane.
  - Explain how dynamic programming is used in the Bellman–Ford shortest path algorithm.
  - Formulate and solve the rod cutting problem using dynamic programming.
  - Describe a memoized solution for the knapsack problem and analyze its complexity.
  - Break down the problem of computing the determinant of a matrix using Laplace expansion and discuss its complexity.
  - Describe how quickselect decomposes the selection problem and its average-case complexity.
  - Use divide-and-conquer to solve the maximum subarray problem and compare with Kadane’s algorithm.
  - Explain how recursion can solve the problem of generating all binary search trees with  $n$  nodes.
  - Design a parallel merge sort algorithm and discuss how the problem is divided among processors.
- 

## 5. Pseudocode Interpretation

### 5.1 Conceptual Notes

Pseudocode expresses algorithms in structured but informal language independent of specific programming syntax. Being able to interpret pseudocode is vital for implementing algorithms in any language, estimating their complexity, and verifying correctness. It uses constructs like loops, conditionals and function calls.

Translating pseudocode to code is like following a recipe: understand each instruction, map variables to data structures, and follow the control flow. Carefully interpreting pseudocode allows one to catch subtle bugs and optimize implementation.

### 5.2 Practice Questions and Solutions

#### 1. Bubble sort pseudocode

**Question:** Interpret the following pseudocode for bubble sort and describe its complexity:

```
for i from 1 to n:
  for j from 1 to n-i:
    if A[j] > A[j+1]:
      swap A[j] and A[j+1]
```



**Solution:** Bubble sort repeatedly steps through the list, compares adjacent items and swaps them if out of order. The outer loop runs  $n$  times; the inner loop runs up to  $n-i$  times. The total number of comparisons is roughly  $n(n-1)/2$ , leading to  $O(n^2)$  time. Space complexity is  $O(1)$ .

## 2. Binary search pseudocode

**Question:** Given pseudocode for binary search:

```
low = 0
high = n-1
while low ≤ high:
    mid = (low + high)/2
    if A[mid] == key: return mid
    if A[mid] < key: low = mid + 1
    else: high = mid - 1
return -1
```

**Solution:** Binary search maintains low and high pointers into the array. Each iteration compares the key with the middle element and discards half of the search space. At most  $\log_2 n$  iterations are needed. The algorithm runs in  $O(\log n)$  time and  $O(1)$  space.

## 3. Insertion sort pseudocode

**Question:** Translate insertion sort pseudocode into Python and state its complexity.

**Solution:** Pseudocode: for  $i$  from 1 to  $n-1$ :  $current = A[i]$ ;  $j = i-1$ ; while  $j \geq 0$  and  $A[j] > current$ :  $A[j+1] = A[j]$ ;  $j = j-1$ ; set  $A[j+1] = current$ . This algorithm inserts  $A[i]$  into the sorted subarray  $A[0 \dots i-1]$ . The worst-case time is  $O(n^2)$ . The Python implementation uses nested loops accordingly.

## 4. Sum of squares pseudocode

**Question:** Given pseudocode to compute the sum of the first  $n$  squares:

```
total = 0
for i from 1 to n:
    total = total + i * i
return total
```

What is the output when  $n = 3$ , and what is the complexity?

**Solution:** For  $n = 3$ , the loop computes  $1^2 + 2^2 + 3^2 = 1 + 4 + 9 = 14$ . The time complexity is  $O(n)$  due to the single loop.

## 5. BFS pseudocode interpretation



**Question:** Interpret pseudocode for Breadth-First Search (BFS): enqueue start; while queue not empty:  $u =$  dequeue; for each neighbor  $v$  of  $u$ : if  $v$  not visited: mark visited and enqueue  $v$ .

**Solution:** BFS visits nodes level by level using a queue. It ensures that the shortest path (in unweighted graphs) is discovered first. The algorithm runs in  $O(V + E)$  time and  $O(V)$  space.

## 6. FizzBuzz pseudocode

**Question:** A pseudocode snippet categorizes an integer  $n$ :

```
if  $n \% 15 == 0$ : print("FizzBuzz")
else if  $n \% 3 == 0$ : print("Fizz")
else if  $n \% 5 == 0$ : print("Buzz")
else: print(n)
```

What is printed when  $n = 20$ ?

**Solution:** The first condition ( $n$  divisible by 15) is false for 20. The second (divisible by 3) is false, and the third (divisible by 5) is true. Thus the output is **"Buzz"**.

## 7. Recursive factorial trace

**Question:** Trace the execution of the following recursive pseudocode for factorial when  $n = 4$ :

```
function fact( $n$ ):
    if  $n == 0$ : return 1
    else: return  $n * \text{fact}(n-1)$ 
```

**Solution:** The call  $\text{fact}(4)$  expands as  $4 * \text{fact}(3)$ ;  $\text{fact}(3) = 3 * \text{fact}(2)$ ;  $\text{fact}(2) = 2 * \text{fact}(1)$ ;  $\text{fact}(1) = 1 * \text{fact}(0)$ ;  $\text{fact}(0) = 1$ . Hence  $\text{fact}(4) = 4 \times 3 \times 2 \times 1 = 24$ .

## 8. Matrix multiplication operations

**Question:** Count the number of multiplications performed by the following pseudocode for matrix multiplication:

```
for  $i$  in  $1..n$ :
    for  $j$  in  $1..n$ :
         $C[i][j] = 0$ 
        for  $k$  in  $1..n$ :
             $C[i][j] = C[i][j] + A[i][k] * B[k][j]$ 
```

**Solution:** There are three nested loops each running  $n$  times. The innermost loop performs one multiplication and one addition per iteration. Thus there are  **$n^3$  multiplications** and  **$n^3$  additions**. The time complexity is  **$O(n^3)$** .



## 9. Dynamic Fibonacci pseudocode

**Question:** Interpret the following dynamic programming pseudocode for Fibonacci numbers and state its complexity:

```
fib[0] = 0; fib[1] = 1
for i from 2 to n:
    fib[i] = fib[i-1] + fib[i-2]
return fib[n]
```

**Solution:** The loop fills an array `fib[]` of size  $n+1$ . Each iteration performs a constant-time addition. The algorithm runs in  **$O(n)$**  time and  **$O(n)$**  space. It avoids the exponential blow-up of naive recursion by reusing previously computed values.

## 10. Primality test pseudocode

**Question:** Given pseudocode for a primality test:

```
isPrime(n):
    if n < 2: return False
    for i from 2 to floor( $\sqrt{n}$ ):
        if n % i == 0: return False
    return True
```

Analyze its complexity.

**Solution:** The loop checks divisibility up to the square root of  $n$ . In the worst case (for prime  $n$ ), it performs  $\sqrt{n} - 1$  divisions. The time complexity is  **$O(\sqrt{n})$** .

## 11. Reverse string using stack

**Question:** Interpret pseudocode for reversing a string using a stack.

**Solution:** Push each character of the string onto a stack. Then pop characters sequentially and append them to a result string. This reverses the order. The algorithm runs in  **$O(n)$**  time and  **$O(n)$**  space.

## 12. Quicksort pseudocode complexity

**Question:** Given pseudocode for quicksort, identify how the pivot is chosen and explain the worst-case complexity.

**Solution:** Classic quicksort selects a pivot (e.g., first element). It partitions the array into elements less than the pivot and those greater. Recursively sort subarrays. If the pivot is always the smallest or largest element, the recurrence  $T(n) = T(n-1) + O(n)$  yields  **$O(n^2)$**  worst-case time. Using random or median-of-three pivots mitigates this.

## 13. Prefix sum pseudocode



**Question:** Translate pseudocode for computing prefix sums into code.

**Solution:** Pseudocode: `prefix[0] = A[0]; for i from 1 to n-1: prefix[i] = prefix[i-1] + A[i]`. The Python code initializes a list and iterates, running in  $O(n)$  time and  $O(n)$  space.

#### 14. Merge two sorted arrays

**Question:** Interpret pseudocode that merges two sorted arrays into a single sorted array.

**Solution:** Maintain two indices  $i$  and  $j$  starting at 0. While both arrays have remaining elements, append the smaller of  $A[i]$  and  $B[j]$  to the result and increment the index. Then append remaining elements from either array. This merge runs in  $O(n + m)$  time.

#### 15. Palindrome detection pseudocode

**Question:** Given pseudocode for detecting a palindrome:

```
for i from 0 to n/2:
    if s[i] != s[n-1-i]: return False
return True
```

interpret its operation.

**Solution:** The algorithm compares characters symmetrically from the start and end of the string. If any pair differs, it returns False; otherwise True. It runs in  $O(n)$  time and  $O(1)$  space.

### 5.3 Unsolved Practice Questions

- Convert pseudocode for selection sort into a high-level language and analyze its complexity.
  - Given pseudocode for computing binomial coefficients using Pascal's triangle, implement it and state the complexity.
  - Interpret pseudocode for DFS and determine the order of node visits on a given graph.
  - Translate pseudocode for computing  $(x^n)$  using exponentiation by squaring into code and discuss efficiency.
  - Analyze pseudocode for a naive string matching algorithm and count its comparisons.
  - Given pseudocode for computing the transitive closure of a graph via the Floyd–Warshall algorithm, discuss its complexity.
  - Interpret pseudocode for topological sorting and explain its purpose.
  - Implement pseudocode for rotating a matrix 90 degrees clockwise and analyze its complexity.
  - Translate pseudocode for counting sort into code and specify conditions under which it is efficient.
  - Interpret pseudocode for calculating the depth of a binary tree and implement it.
-



## 6. Recursion Basics

### 6.1 Conceptual Notes

Recursion is a method where a function calls itself to solve subproblems until reaching a base case. Each call adds a frame to the call stack. A well-formed recursive algorithm must define a base case that terminates the recursion and must reduce the problem size at each step. Real-world analogies include Russian nesting dolls and tasks like computing factorials or traversing tree structures. Recursion is natural for problems that exhibit self-similarity, but deep recursion can lead to stack overflow. Understanding recursion helps in designing divide-and-conquer algorithms, backtracking, and dynamic programming.

### 6.2 Practice Questions and Solutions

#### 1. Recursive factorial

**Question:** Write a recursive function to compute  $\text{factorial}(n)$  and state its complexity.

**Solution:** Define  $\text{factorial}(n) = 1$  if  $n \leq 1$ ; else  $n \times \text{factorial}(n-1)$ . Each call reduces  $n$  by 1 until reaching the base case. This makes  $n$  recursive calls, yielding  $O(n)$  time complexity and  $O(n)$  space on the call stack.

#### 2. Countdown recursion

**Question:** Using recursion, print all integers from  $n$  down to 1.

**Solution:** Base case: if  $n \leq 0$ , return. Otherwise, print  $n$  and call the function with  $n-1$ . This prints numbers in descending order. The complexity is  $O(n)$  time and  $O(n)$  space for the call stack.

#### 3. Fibonacci methods

**Question:** Compute the  $n$ th Fibonacci number using both naive recursion and dynamic programming and compare complexities.

**Solution:** Naive recursion defines  $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$  with base cases  $\text{fib}(0)=0$ ,  $\text{fib}(1)=1$ . This leads to exponential time complexity  $O(\phi^n)$ . Dynamic programming (memoization or tabulation) stores previously computed values and runs in  $O(n)$  time and space.

#### 4. Reverse string recursively

**Question:** Reverse a string recursively.

**Solution:** Base case: if the string is empty or length 1, return it. Otherwise return  $\text{reverse}(s[1:]) + s[0]$ . This builds the reversed string in  $O(n)$  time and  $O(n)$  space.

#### 5. Recursive binary search



**Question:** Implement a recursive binary search.

**Solution:** Function `binary_search(arr, low, high, key)`: if `low > high` return `-1`; compute `mid`; if `arr[mid] == key` return `mid`; if `arr[mid] < key` call on the right half; else call on the left half. Each call halves the problem size, giving  $O(\log n)$  time and  $O(\log n)$  space due to recursion.

## 6. Tower of Hanoi

**Question:** Solve the Tower of Hanoi problem recursively for  $n$  disks. State the number of moves.

**Solution:** To move  $n$  disks from peg A to C using B: recursively move  $n-1$  disks from A to B, move the largest disk from A to C, then move  $n-1$  disks from B to C. The number of moves satisfies  $T(n) = 2T(n-1) + 1$ , giving  $T(n) = 2^n - 1$ .

## 7. In-order traversal

**Question:** Perform in-order traversal of a binary tree recursively.

**Solution:** Define `inorder(node)`: if `node` is `None` return; call `inorder(node.left)`; process `node`; call `inorder(node.right)`. This visits nodes in sorted order for a BST. Time complexity is  $O(n)$  and space complexity is  $O(h)$  where  $h$  is the tree height.

## 8. Maximum element via recursion

**Question:** Find the maximum element of an array using recursion.

**Solution:** Define `recursive_max(arr, n)`: if `n == 1` return `arr[0]`; recursively find the max of the first  $n-1$  elements; return `max(arr[n-1], result)`. This runs in  $O(n)$  time and  $O(n)$  space.

## 9. Sum of digits recursively

**Question:** Compute the sum of digits of an integer  $n$  recursively.

**Solution:** Base case: if  $n < 10$  return  $n$ . Otherwise return  $(n \% 10) + \text{sum\_digits}(n // 10)$ . Each call removes one digit. Complexity is  $O(d)$  where  $d$  is the number of digits.

## 10. Exponentiation by squaring

**Question:** Implement exponentiation ( $x^n$ ) using recursion and exponentiation by squaring.

**Solution:** If  $n == 0$  return 1. If  $n$  is even, compute `half = pow(x, n/2)`; return `half * half`. If  $n$  is odd, return `x * pow(x, n-1)`. This reduces the exponent quickly, giving  $O(\log n)$  time and  $O(\log n)$  space.

## 11. Merge sort recursively

**Question:** Use recursion to implement merge sort and state its complexity.



**Solution:** Merge sort recursively splits the array into halves, sorts each half, and merges them. The recurrence  $T(n) = 2T(n/2) + n$  yields  $O(n \log n)$  time. Space complexity is  $O(n)$  for the auxiliary array used in merging and  $O(\log n)$  for the call stack.

## 12. Recursive GCD

**Question:** Compute the greatest common divisor (GCD) of two numbers using Euclid's algorithm recursively.

**Solution:** Base case:  $\text{gcd}(a, 0) = a$ . Otherwise return  $\text{gcd}(b, a \bmod b)$ . Each call reduces the second argument, yielding  $O(\log \min(a, b))$  time and  $O(\log \min(a, b))$  space.

## 13. Climbing stairs

**Question:** Count the number of ways to climb a staircase with  $n$  steps, taking 1 or 2 steps at a time.

**Solution:** Let  $\text{ways}(n)$  be the count. Base cases:  $\text{ways}(0)=1$ ,  $\text{ways}(1)=1$ . Recursive relation:  $\text{ways}(n) = \text{ways}(n-1) + \text{ways}(n-2)$ . This is the Fibonacci sequence. A naive recursive solution has exponential time, whereas memoization yields  $O(n)$  time.

## 14. Linked list palindrome

**Question:** Use recursion to determine whether a linked list is a palindrome.

**Solution:** Define a helper that recurses to the end of the list and compares symmetric nodes while unwinding. Alternatively, reverse the second half of the list and compare. Both solutions run in  $O(n)$  time; the recursive solution uses  $O(n)$  space on the call stack.

## 15. Generate power set

**Question:** Generate all subsets (the power set) of a set recursively.

**Solution:** For a set  $S$ , define  $\text{subsets}(S) = \text{subsets}(S \text{ without first element})$  plus copies of those subsets with the first element added. Base case: subsets of an empty set is  $\{\emptyset\}$ . This yields  $(2^n)$  subsets and runs in  $O(n \cdot 2^n)$  time.

## 6.3 Unsolved Practice Questions

- Write a recursive function to compute the  $n$ th triangular number (sum of integers 1 to  $n$ ).
- Use recursion to convert a decimal number to binary representation.
- Design a recursive algorithm to calculate the power of a number with negative exponents.
- Implement post-order traversal of a binary tree using recursion.
- Solve the problem of generating all combinations of balanced parentheses of length  $2n$ .
- Implement a recursive function to count the number of nodes in a binary tree.
- Use recursion to find the first index of a target value in an array.
- Write a recursive function to compute the binomial coefficient  $(C(n, k))$ .
- Generate permutations of a string recursively.



# Data Interpretation & Logical Reasoning

## 1. Essential Concepts & Terminology

### 1.1 Percentage

**Definition:** A number or ratio expressed as a fraction of 100, used to compare proportions and express changes.

**Formula:**  $(\text{Part} / \text{Whole}) \times 100$

**Key Applications:** - Profit and loss analysis - Market share calculation - Percentage increase/decrease in sales, production, etc.

**Example:** If a company's revenue increased from ₹500 crore to ₹650 crore, the percentage increase =  $((650-500)/500) \times 100 = 30\%$

### 1.2 Ratio

**Definition:** A mathematical comparison between two quantities showing their relative sizes or proportions.

**Formula:** A:B or A/B

**Key Applications:** - Profit distribution among partners - Resource allocation - Comparison problems

**Example:** If expenses are distributed in ratio 3:2:1 among three departments with total budget ₹600,000, allocations are ₹300,000, ₹200,000, and ₹100,000.

### 1.3 Average (Mean)

**Definition:** The sum of all values in a dataset divided by the total number of values, representing central tendency.

**Formula:**  $(\text{Sum of values}) / (\text{Number of values})$

**Key Applications:** - Finding central tendency of data - Average performance calculation - Comparing different datasets

**Example:** Average sales over 5 months (100, 120, 110, 130, 140) =  $(100+120+110+130+140)/5 = 120$  units

### 1.4 Trend

**Definition:** The general direction or tendency shown by data over a period of time.

**Types of Trends:** - **Uptrend:** Consistent increase over time - **Downtrend:** Consistent decrease over time - **Flat/Stable:** Minimal change - **Cyclical:** Regular patterns of ups and downs - **Volatile:** Irregular fluctuations

**Key Applications:** - Sales growth analysis - Performance tracking over time - Forecasting future patterns

### 1.5 Data Point

**Definition:** An individual value or observation within a dataset, representing a specific measurement or fact.



**Identification Methods:** - Visual scan for highest/lowest values - Numerical comparison - Pattern recognition

**Key Applications:** - Extreme value identification (maximum/minimum) - Outlier detection - Pattern and anomaly recognition

## 1.6 Cross-Reference

**Definition:** The process of comparing or linking data from multiple sources, tables, or charts to derive comprehensive insights.

**Key Applications:** - Multi-table data interpretation - Connecting information across different representations - Comprehensive analysis requiring multiple data sources

**Example:** Combining revenue data from Table A with expense data from Chart B to calculate profit margins.

## 1.7 Pattern

**Definition:** A recurring sequence, arrangement, or rule that governs how elements are organized or related.

**Pattern Types:** - **Numerical Patterns:** Arithmetic, geometric, Fibonacci sequences - **Alphabetical Patterns:** Letter sequences, position-based - **Visual Patterns:** Shape, size, orientation changes - **Logical Patterns:** Conditional or rule-based sequences

**Key Applications:** - Number series completion - Matrix and analogy problems - Sequential reasoning

## 1.8 Logical Deduction

**Definition:** The process of deriving specific conclusions from general statements or premises using logical reasoning.

**Methods:** - **Syllogism:** Two premises leading to a conclusion - **Venn Diagram Method:** Visual representation of logical relationships - **Direct Inference:** Immediate conclusions from single statements

**Key Applications:** - Statement-conclusion problems - Critical reasoning - Argument evaluation

## 1.9 Assumption

**Definition:** An unstated belief, supposition, or premise that underlies a statement or argument and must be true for the statement to make sense.

**Role:** Bridges the gap between evidence and conclusion; must be valid for the argument to hold.

**Key Applications:** - Identifying unstated premises in arguments - Evaluating argument validity - Critical analysis of statements

**Example:** Statement: "We should invest in Company X because their revenue grew 20% last year."  
Assumption: Revenue growth is a reliable indicator of investment worthiness.

## 1.10 Cause and Effect

**Definition:** - **Cause:** The reason or action that produces an event or outcome - **Effect:** The result or consequence of an action or event



**Relationship Types:** - **Direct Causation:** A directly causes B - **Common Cause:** C causes both A and B - **Independent Causes:** A and B occur independently

**Key Applications:** - Analyzing cause-effect relationships - Distinguishing correlation from causation - Problem-solving and decision-making

---

## 2. Conceptual Framework & Problem-Solving Methodology

### 2.1 Data Interpretation

#### Bar Charts

**Description:** Visual representation using rectangular bars to show quantities, frequencies, or comparisons across categories.

**Key Analysis Skills:** - Identifying maximum/minimum values - Calculating percentage changes - Comparing multiple categories - Finding averages and totals

**Step-by-Step Method:** 1. Read the title and axes carefully 2. Identify units of measurement 3. Scan for highest/lowest bars quickly 4. Calculate required values systematically 5. Verify calculations before finalizing

#### Worked Example:

**Question:** A bar chart shows monthly sales (in lakhs) for 6 months: Jan(50), Feb(60), Mar(55), Apr(70), May(65), Jun(80). Find the average monthly sales and percentage increase from Jan to Jun.

**Solution:** - Average =  $(50+60+55+70+65+80)/6 = 380/6 = 63.33$  lakhs - Percentage increase =  $((80-50)/50) \times 100 = 60\%$

#### Line Graphs

**Description:** Connected data points showing trends, changes, and patterns over continuous intervals (usually time).

**Key Analysis Skills:** - Identifying trends (upward, downward, stable) - Finding rate of change - Comparing multiple lines - Identifying intersection points

**Step-by-Step Method:** 1. Observe overall trend direction 2. Identify key turning points 3. Calculate slopes for rate analysis 4. Compare parallel trends if multiple lines 5. Focus on specific time periods as required

#### Pie Charts

**Description:** Circular chart divided into sectors representing proportions or percentages of a whole.

**Key Analysis Skills:** - Understanding part-to-whole relationships - Calculating actual values from percentages - Comparing sector sizes - Finding combined percentages

**Step-by-Step Method:** 1. Note the total value represented 2. Identify percentage of each sector 3. Calculate actual values:  $(\text{Percentage} \times \text{Total})/100$  4. Compare sectors for ranking 5. Combine sectors as needed for answers



## Tables

**Description:** Organized data in rows and columns showing multiple variables and their relationships.

**Key Analysis Skills:** - Cross-referencing multiple data points - Row and column calculations - Finding totals and subtotals - Identifying patterns across categories

**Step-by-Step Method:** 1. Understand row and column headers 2. Identify relevant data cells 3. Perform required calculations systematically 4. Check for missing data or footnotes 5. Verify by recalculating critical values

### *Data Analysis Essential Skills*

Skill	Technique
Approximation	Round numbers for quick estimation
Comparison	Direct ratio or percentage comparison
Verification	Cross-check with multiple methods
Pattern Recognition	Identify recurring trends in data

## 2.2 Logical & Analytical Reasoning

### 1. Pattern Recognition

**Description:** Identifying underlying rules or sequences in numerical, alphabetical, or visual series.

**Common Pattern Types:** - **Arithmetic Progression:** Constant difference (e.g., 2, 5, 8, 11...) - **Geometric Progression:** Constant ratio (e.g., 3, 6, 12, 24...) - **Squared Series:** Based on squares (e.g., 1, 4, 9, 16...) - **Prime Numbers:** 2, 3, 5, 7, 11, 13... - **Fibonacci:** Each term = sum of previous two (e.g., 1, 1, 2, 3, 5, 8...)

**Step-by-Step Method:** 1. Calculate differences between consecutive terms 2. Check for constant difference (arithmetic) 3. Calculate ratios if not arithmetic (geometric) 4. Look for alternating patterns or dual sequences 5. Test the identified pattern on all given terms

#### **Worked Example:**

**Question:** Find the next number in the series: 3, 7, 15, 31, 63, ?

**Solution:** - Pattern: Each term = (Previous term  $\times$  2) + 1 -  $3 \rightarrow 7$  ( $3 \times 2 + 1$ ),  $7 \rightarrow 15$  ( $7 \times 2 + 1$ ),  $15 \rightarrow 31$  ( $15 \times 2 + 1$ ),  $31 \rightarrow 63$  ( $31 \times 2 + 1$ ) - Next term =  $63 \times 2 + 1 = 127$

### 2. Sequential Reasoning

**Description:** Arranging items or events in a logical order based on given conditions or rules.

**Key Techniques:** - Creating position charts or diagrams - Elimination method for impossible arrangements - Testing each condition systematically - Working with definite statements first



**Step-by-Step Method:** 1. List all given conditions clearly 2. Identify definite/fixed positions first 3. Use process of elimination 4. Draw diagrams for complex arrangements 5. Verify final sequence against all conditions

### 3. Critical Thinking

**Description:** Analyzing arguments, statements, and conclusions to evaluate their validity and logical soundness.

**Key Skills:** - Identifying assumptions in arguments - Distinguishing between facts and opinions - Evaluating strength of conclusions - Detecting logical fallacies

**Common Question Types:** - Assumptions - Inferences - Strengthen/Weaken - Paradox Resolution

### 4. Abstract Reasoning

**Description:** Understanding and manipulating abstract concepts, symbols, and visual patterns without relying on language or concrete examples.

**Key Applications:** - Matrix completion problems - Figure analogies - Symbol-based sequences - Spatial reasoning

**Step-by-Step Method:** 1. Identify all visual elements (shapes, lines, colors, positions) 2. Look for transformations (rotation, reflection, scaling) 3. Check for progressive changes across sequence 4. Test hypothesis on multiple examples 5. Select answer matching the identified pattern

### 5. Logical Deduction

**Description:** Drawing specific conclusions from general statements using formal logical rules.

#### Syllogism Basics:

Premise 1 + Premise 2  $\rightarrow$  Conclusion

**Valid Conclusion Rules:** - All A are B + All B are C  $\rightarrow$  All A are C (Transitivity) - No A are B + All C are A  $\rightarrow$  No C are B - Some A are B + All B are C  $\rightarrow$  Some A are C

#### Worked Example:

**Premises:** 1. All engineers are logical thinkers. 2. Some logical thinkers are creative.

**Question:** Which conclusion follows?

**Analysis:** - Valid: Some engineers MAY be creative (possible but not certain) - Invalid: All engineers are creative (too strong) - Invalid: No engineer is creative (contradicts possibility)

**Correct Conclusion:** No definite conclusion can be drawn about engineers and creativity from these premises alone.



### 3. Solved Examples With Detailed Explanations

#### Example 1: Data Interpretation - Bar Chart (EASY)

**Question:** A bar chart shows the number of laptops sold by a company over 5 years: 2019(200), 2020(250), 2021(300), 2022(280), 2023(350). What was the percentage increase in sales from 2019 to 2023?

**Step-by-Step Solution:**

**Step 1:** Identify initial and final values - Initial (2019) = 200 laptops - Final (2023) = 350 laptops

**Step 2:** Calculate absolute increase - Increase =  $350 - 200 = 150$  laptops

**Step 3:** Calculate percentage increase - Percentage =  $(\text{Increase} / \text{Initial}) \times 100 = (150 / 200) \times 100 = 0.75 \times 100 = 75\%$

**Answer:** 75%

#### Example 2: Data Interpretation - Table Analysis (MEDIUM)

**Question:** A table shows quarterly revenue (in crores) for three products:

Product	Q1	Q2	Q3	Q4
A	50	60	55	65
B	40	45	50	55
C	70	65	75	80

What is the average quarterly revenue for Product B, and which product had the highest total annual revenue?

**Step-by-Step Solution:**

**Part 1: Average for Product B** - Q1 to Q4 values: 40, 45, 50, 55 - Average =  $(40 + 45 + 50 + 55) / 4 = 190 / 4 = 47.5$  crores

**Part 2: Highest total annual revenue** - Product A total =  $50 + 60 + 55 + 65 = 230$  crores - Product B total =  $40 + 45 + 50 + 55 = 190$  crores - Product C total =  $70 + 65 + 75 + 80 = 290$  crores (**Highest**)

**Answer:** Average for B = 47.5 crores; Product C had highest revenue

#### Example 3: Number Series Pattern (MEDIUM)

**Question:** Find the missing number in the series: 5, 11, 23, 47, 95, ?

**Step-by-Step Solution:**

**Step 1:** Calculate differences between consecutive terms -  $11 - 5 = 6$  -  $23 - 11 = 12$  -  $47 - 23 = 24$  -  $95 - 47 = 48$

**Step 2:** Identify pattern in differences - 6, 12, 24, 48 → Each difference doubles ( $\times 2$ )

**Step 3:** Find next difference - Next difference =  $48 \times 2 = 96$

**Step 4:** Calculate missing number - Missing number =  $95 + 96 = 191$



**Answer: 191**

**Alternative Pattern Recognition:** Each term = (Previous term  $\times$  2) + 1 -  $5 \times 2 + 1 = 11 \checkmark$  -  $11 \times 2 + 1 = 23 \checkmark$  -  $23 \times 2 + 1 = 47 \checkmark$  -  $47 \times 2 + 1 = 95 \checkmark$  -  $95 \times 2 + 1 = 191 \checkmark$

#### Example 4: Logical Deduction - Syllogism (MEDIUM)

**Premises:** 1. All scientists are researchers. 2. Some researchers are professors.

**Question:** Which of the following conclusions is valid?

- A) All scientists are professors
- B) Some scientists are professors
- C) No scientist is a professor
- D) None of the above

**Step-by-Step Solution:**

**Step 1:** Analyze Premise 1 - All scientists  $\subset$  researchers (Scientists are subset of researchers)

**Step 2:** Analyze Premise 2 - Some researchers  $\cap$  professors (Partial overlap)

**Step 3:** Venn Diagram Analysis - Scientists could be in the overlapping region (researchers  $\cap$  professors) OR outside it. Both scenarios are possible with given premises.

**Step 4:** Evaluate options - A) All scientists are professors  $\rightarrow$  Too strong, NOT necessarily true  $\times$  - B) Some scientists are professors  $\rightarrow$  POSSIBLE but not certain  $\times$  - C) No scientist is a professor  $\rightarrow$  POSSIBLE but not certain  $\times$  - D) None of the above  $\rightarrow$  Correct, no definite conclusion  $\checkmark$

**Answer: D) None of the above**

#### Example 5: Percentage Calculation - Multi-Step (HARD)

**Question:** A product's price increased by 25% in Year 1, then decreased by 20% in Year 2. If the final price is ₹600, what was the original price?

**Step-by-Step Solution:**

**Step 1:** Define original price - Let original price = P

**Step 2:** Calculate price after Year 1 increase (25%) - Price after Year 1 =  $P + 0.25P = 1.25P$

**Step 3:** Calculate price after Year 2 decrease (20%) - Price after Year 2 =  $1.25P - (0.20 \times 1.25P) = 1.25P - 0.25P = 1.00P$

**Step 4:** Set up equation -  $1.00P = 600$  -  $P = ₹600$

**Answer: ₹600**

**Key Insight:** A 25% increase followed by a 20% decrease (of the new amount) brings the price back to original! This is because  $20\% \text{ of } 125 = 25$ .

**Verification:** - Original: ₹600 - After +25%: ₹750 - After -20% of ₹750: ₹750 - ₹150 = ₹600  $\checkmark$



### Example 6: Ratio and Proportion (MEDIUM)

**Question:** Three partners A, B, and C invest in a business in the ratio 3:4:5. If the total profit is ₹84,000, how much does partner B receive?

**Step-by-Step Solution:**

**Step 1:** Identify ratio components - A:B:C = 3:4:5 - Total ratio parts = 3 + 4 + 5 = 12

**Step 2:** Calculate value of one part - 12 parts = ₹84,000 - 1 part = ₹84,000 / 12 = ₹7,000

**Step 3:** Calculate B's share - B's share = 4 parts - = 4 × ₹7,000 - = ₹28,000

**Answer:** ₹28,000

**Verification:** - A = 3 × 7,000 = ₹21,000 - B = 4 × 7,000 = ₹28,000 - C = 5 × 7,000 = ₹35,000 - Total = 21,000 + 28,000 + 35,000 = ₹84,000 ✓

### Example 7: Cause and Effect Analysis (EASY)

**Statement I:** The government increased fuel taxes by 15%. **Statement II:** Transportation costs for goods rose by 10%.

**Question:** What is the relationship between the two statements?

- A) Statement I is the cause and Statement II is its effect
- B) Statement II is the cause and Statement I is its effect
- C) Both are independent causes
- D) Both are effects of a common cause

**Step-by-Step Solution:**

**Step 1:** Identify temporal sequence - Tax increase (Statement I) happens first

**Step 2:** Analyze logical connection - Fuel taxes directly impact transportation costs because: - Transport requires fuel - Higher fuel taxes → higher fuel prices → higher transport costs

**Step 3:** Determine relationship - Statement I (tax increase) causes Statement II (cost rise)

**Answer:** A) Statement I is the cause and Statement II is its effect

**Key Reasoning:** Direct causal relationship exists because transportation inherently requires fuel, making fuel cost changes directly affect transport costs.

### Example 8: Data Interpretation - Pie Chart (HARD)

**Question:** A pie chart shows the distribution of a company's ₹500 crore annual budget: - Production: 35% - Marketing: 25% - R&D: 20% - Administration: 15% - Others: 5%

If the company decides to increase the R&D budget by ₹30 crore while keeping other absolute amounts constant, what will be the new percentage allocation for R&D?

**Step-by-Step Solution:**

**Step 1:** Calculate original R&D amount - Original R&D = 20% of ₹500 crore - = 0.20 × 500 = ₹100 crore



**Step 2:** Calculate new R&D amount - New R&D = ₹100 crore + ₹30 crore = ₹130 crore

**Step 3:** Calculate new total budget - Original total = ₹500 crore - Increase = ₹30 crore - New total = ₹530 crore

**Step 4:** Calculate new R&D percentage - New R&D percentage =  $(130 / 530) \times 100 = 0.2453 \times 100 = 24.53\%$  (approximately 24.5%)

**Answer: 24.53% or approximately 24.5%**

**Verification:** New percentages should account for ₹530 crore total: - Production:  $(175/530) \times 100 = 33.02\%$  - Marketing:  $(125/530) \times 100 = 23.58\%$  - R&D:  $(130/530) \times 100 = 24.53\%$  ✓ - (All percentages decrease slightly except R&D)

---

## 4. Practice Questions & Self-Assessment

### SET A: Foundation Level (Easy)

**Time Allocation:** 4 minutes | **Target Score:** 3-4 correct

**Q1.** In a company, 60% of employees are male and 40% are female. If there are 180 female employees, how many total employees are there?

- A) 300
- B) 360
- C) 450
- D) 500

**Q2.** Find the next number in the series: 2, 6, 12, 20, 30, ?

- A) 40
- B) 42
- C) 44
- D) 48

**Q3.** If the average of five numbers is 45, and four of them are 40, 42, 48, and 50, what is the fifth number?

- A) 43
- B) 45
- C) 47
- D) 49

**Q4.** A product costs ₹800 after a 20% discount. What was the original price?

- A) ₹960
- B) ₹1,000
- C) ₹1,200
- D) ₹1,500



### SET B: Intermediate Level (Medium)

**Time Allocation:** 6 minutes | **Target Score:** 3-4 correct

**Q5.** A table shows sales (in thousands) for three products over 4 months:

Product	Jan	Feb	Mar	Apr
X	50	55	60	65
Y	40	50	45	55
Z	60	58	62	64

What is the percentage increase in Product Y's sales from January to April?

- A) 25%
- B) 30%
- C) 35%
- D) 37.5%

**Q6.** Three friends divide ₹75,000 in the ratio 2:3:5. What is the difference between the highest and lowest share?

- A) ₹15,000
- B) ₹20,000
- C) ₹22,500
- D) ₹30,000

**Q7. Premises:** - All managers are leaders. - No leader is a follower.

**Conclusion:** No manager is a follower. Is this conclusion valid?

- A) Yes, definitely valid
- B) No, definitely invalid
- C) Possibly valid
- D) Cannot be determined

**Q8.** Find the missing number: 7, 14, 28, 35, 70, 77, ?

- A) 140
- B) 144
- C) 154
- D) 164

**Q9.** If A is 50% more than B, and B is 20% less than C, then A is what percentage of C?

- A) 110%
- B) 115%
- C) 120%
- D) 125%



### SET C: Advanced Level (Hard)

**Time Allocation:** 8 minutes | **Target Score:** 2-3 correct

**Q10.** A company's profit increased by 20% in Year 1 and by 25% in Year 2. If the profit in Year 2 was ₹37.5 lakhs, what was the profit before Year 1?

- A) ₹20 lakhs
- B) ₹22.5 lakhs
- C) ₹25 lakhs
- D) ₹30 lakhs

**Q11.** A pie chart shows budget allocation: Department A gets 30%, B gets 25%, C gets 20%, D gets 15%, and E gets 10%. If Department C's allocation is increased by ₹40 lakhs while total budget increases to ₹620 lakhs, what was the original total budget?

- A) ₹500 lakhs
- B) ₹550 lakhs
- C) ₹580 lakhs
- D) ₹600 lakhs

**Q12.** In a sequence, each term after the first two is the sum of the two preceding terms. If the 5th term is 29 and the 7th term is 76, what is the 3rd term?

- A) 7
- B) 11
- C) 13
- D) 18

**Q13. Statement:** "Since Company X's market share increased from 15% to 25%, their profits must have doubled."

**Question:** Which assumption is implicit in this statement?

- A) Market share directly correlates with profit
- B) The total market size remained constant
- C) Profit margins remained unchanged
- D) All of the above

---

### Answer Key & Detailed Explanations

#### SET A: Foundation Level - Answers

**Q1. Answer: C) 450**

**Explanation:** If  $40\% = 180$  employees, then  $100\% = (180/40) \times 100 = 450$  employees

**Concept:** Percentage to total conversion

**Q2. Answer: B) 42**



**Explanation:** Pattern: Differences are 4, 6, 8, 10, 12... (increasing by 2)

$$2 + 4 = 6, 6 + 6 = 12, 12 + 8 = 20, 20 + 10 = 30, 30 + 12 = 42$$

**Concept:** Arithmetic sequence with increasing differences

**Q3. Answer: B) 45**

**Explanation:** - Sum of 5 numbers =  $45 \times 5 = 225$  - Sum of 4 known numbers =  $40 + 42 + 48 + 50 = 180$  - Fifth number =  $225 - 180 = 45$

**Concept:** Average and missing value calculation

**Q4. Answer: B) ₹1,000**

**Explanation:** If 20% discount  $\rightarrow$  price is 80% of original.  $80\% = ₹800$ , so  $100\% = (800/80) \times 100 = ₹1,000$

**Concept:** Reverse percentage calculation

### SET B: Intermediate Level - Answers

**Q5. Answer: D) 37.5%**

**Explanation:** - Product Y: Jan = 40, Apr = 55 - Increase =  $55 - 40 = 15$  - Percentage =  $(15/40) \times 100 = 37.5\%$

**Concept:** Percentage increase from table data

**Q6. Answer: C) ₹22,500**

**Explanation:** - Ratio 2:3:5, total parts = 10 - One part =  $75,000/10 = ₹7,500$  - Lowest (2 parts) = ₹15,000, Highest (5 parts) = ₹37,500 - Difference =  $37,500 - 15,000 = ₹22,500$

**Concept:** Ratio distribution and difference

**Q7. Answer: A) Yes, definitely valid**

**Explanation:** - All managers  $\subset$  leaders, and leaders  $\cap$  followers =  $\emptyset$  - Therefore, managers  $\cap$  followers =  $\emptyset$  (No manager is a follower)

**Concept:** Syllogistic reasoning with transitivity

**Q8. Answer: C) 154**

**Explanation:** Pattern: Alternating operations -  $7 \times 2 = 14, 14 \times 2 = 28, 28 + 7 = 35, 35 \times 2 = 70, 70 + 7 = 77, 77 \times 2 = 154$

**Concept:** Dual operation sequence

**Q9. Answer: C) 120%**

**Explanation:** - Let C = 100 - B = 80 (20% less than C) - A = 120 (50% more than B:  $80 + 40$ ) -  $A/C = 120/100 = 120\%$

**Concept:** Multi-step percentage relationships



### SET C: Advanced Level - Answers

**Q10. Answer: C) ₹25 lakhs**

**Explanation:** Working backward: - Year 2 profit = ₹37.5 lakhs = Year 1 profit  $\times$  1.25 - Year 1 profit = 37.5 / 1.25 = ₹30 lakhs - Year 0 profit = 30 / 1.20 = ₹25 lakhs

**Concept:** Reverse percentage calculation with multiple steps

**Q11. Answer: D) ₹600 lakhs**

**Explanation:** Let original budget = B. Original C allocation = 0.20B. New C allocation = 0.20B + 40. New total = 620. Solving: B = ₹600 lakhs

**Concept:** Complex percentage with changing totals

**Q12. Answer: B) 11**

**Explanation:** Fibonacci-type sequence: a, b, a+b, a+2b, 2a+3b, 3a+5b, 5a+8b - 5th term: 2a+3b = 29 - 7th term: 5a+8b = 76 - Solving: b = 7, a = 4 - 3rd term = a + b = 4 + 7 = 11

**Concept:** Fibonacci sequence with algebraic solution

**Q13. Answer: D) All of the above**

**Explanation:** For profits to double just because market share increased: - Assumes market share correlates with profit (A) - Assumes market size didn't change, otherwise percentages misleading (B) - Assumes profit margins stayed same (C)

All three assumptions are necessary for the conclusion.

**Concept:** Multiple implicit assumptions in arguments

---