Relationship Between JVM, JRE, and JDK:

1. **JDK (Java Development Kit)**
    - Includes **JRE** + **Development Tools** (like javac, javadoc, etc.).
2. **JRE (Java Runtime Environment)**
    - Includes **JVM** + **Core Java Libraries** (required for running Java applications).
3. **JVM (Java Virtual Machine)**
    - The JVM is **a process that runs Java bytecode** and converts it into machine code.
    - It is **part of the JRE** and cannot function alone.

Note:

Unlike the JRE, which is **a software package** that you install, the **JVM is an abstract specification** that different vendors implement (like Oracle JVM, OpenJDK JVM, etc.).

How a java program is executed :

When you run a Java program (java MyProgram):

- The **JRE starts the JVM** as a process.
- The JVM **loads** the compiled Java **bytecode (.class files)**.
- The JVM **interprets or compiles** the bytecode into **native machine code** for execution.

Once execution finishes, the **JVM process terminates**.

Note:

The **JVM is part of the JRE**, but it is not something you install separately.

The **JVM is a virtual process** responsible for executing Java bytecode.

The **JRE acts as a container** that provides the **JVM + Java libraries** needed for execution.

✅ **JVM is "spun up" (started) by the JRE when running a Java program**

This is why Java is **platform-independent**—because different operating systems have their own **JVM implementations** inside the JRE.

Difference Between PATH and CLASSPATH in Java

Feature	PATH	CLASSPATH
Purpose	Specifies the location of executables like `javac` and `java`	Specifies the location of Java class files, JARs, and resources
Used By	Operating system	Java compiler (`javac`) and JVM (`java`)
Default Value	Includes system paths like `/usr/bin` (Linux/macOS) or `C:\Windows\System32` (Windows)	Includes standard Java libraries (`rt.jar` in older versions)
Set Using	Environment variable (`PATH`)	Environment variable (`CLASSPATH`) or `-classpath` / `-cp` option
Typical Value	Points to `bin` folder of JDK (`C:\Program Files\Java\jdk\bin`)	Points to directories or JAR files containing Java classes
Example	`C:\Program Files\Java\jdk\bin`	`.;C:\myproject\lib\someLibrary.jar`

Key Notes:

- `PATH` ensures commands like `java` and `javac` work from anywhere in the terminal.
- `CLASSPATH` ensures the JVM finds user-defined classes and external libraries.

Introduction to Programming

Variables and Data Types in Java

Variables : Variables are containers for storing values / data/ information.

Variable declaration:

To create / declare a variable, we must tell Java the variable's data type followed by the variable's name:

```
String name;
```

Variable initialization:

We can initialize a variable to an initial value in the same line as it is created using the assignment operator (=).

```
String name = "Ramana";
```

Java has two categories of data types:

1. **Primitive Data Types** (Built-in)
2. **Non-Primitive Data Types** (Reference types)

1. Primitive Data Types (8 Types)

These are basic types that store values directly in memory.

Data Type	Size	Default Value	Description
`byte`	1 byte	`0`	Stores integers from **-128 to 127**
`short`	2 bytes	`0`	Stores integers from **-32,768 to 32,767**
`int`	4 bytes	`0`	Stores integers from **-2^{31} to $2^{31}-1$**
`long`	8 bytes	`0L`	Stores large integers from **-2^{63} to $2^{63}-1$**
`float`	4 bytes	`0.0f`	Stores fractional numbers with **7 decimal digits precision**
`double`	8 bytes	`0.0d`	Stores fractional numbers with **15 decimal digits precision**
`char`	2 bytes	`\u0000`	Stores a **single character** (Unicode)
`boolean`	1 bit	`false`	Stores **true or false**

2. Non-Primitive Data Types (Reference Types)

These store memory **addresses** instead of actual values.

Data Type	Description
`String`	Stores a sequence of characters
`Array`	Stores multiple values of the same type
`Class`	A blueprint for creating objects
`Interface`	Defines abstract methods that a class must implement
`Enum`	Defines a fixed set of constants

3. Wrapper Classes for Primitives

Primitive types have **wrapper classes** in `java.lang` package:

Primitive	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Key Differences: Primitive vs. Non-Primitive

Feature	Primitive Data Type	Non-Primitive Data Type
Stores	Actual value	Reference (memory address)
Performance	Fast	Slower (due to object overhead)
Default Value	Has a default	Null (if not initialized)
Can be null?	❌ No	✅ Yes

Important notice:

We almost never want to create a variable without initializing it! It's a bad coding practice and can create a lot of problems.

Variable Naming Rules in Java

In Java, we use the camelCase convention to name our variables. It implies separating different words with capital letters:

```
String companyName = "Apple";

int foundingYear = 1976;

boolean isUnicorn = true;

int numberOfEmployees = 137000;
```

Note: Variable names must consist ONLY of letters, numbers and underscores and they must start with a letter or underscore.

Constants in Java

There is a special type of variable in Java, which is called constant or final.

A constant is a variable that we do not want to change the value of during the whole program.

Creation:

You create a constant variable just like you create a normal one, but adding the keyword **final** before the variable's data type:

```
final String ourName = "Ramana";
```

System.out.println(ourName); // Output: Ramana

Reassignment not allowed:

However, a final variable cannot be reassigned because it is constant. If you try to reassign a final variable, you'll get an error.

Ex:

final String ourName = "Ramana";

ourName = "Krish"; // Here is the problem

Memory Management in Java

Java manages memory automatically using Garbage Collection and the JVM Memory Model.

Java divides memory into **two main areas**:

1. **Stack Memory** (For primitive data & method calls)
2. **Heap Memory** (For objects & classes)

Garbage Collection (Automatic Memory Management):

Java **automatically reclaims unused memory** using the **Garbage Collector (GC)**.

- **Objects with no references** are considered **garbage**.
- GC **removes unreachable objects** to free memory.

Feature	Stack Memory	Heap Memory
Stores	Local variables, method calls	Objects, instance variables
Size	Small	Large
Access Speed	Faster	Slower
Managed By	JVM (automatically cleared)	Garbage Collector
Lifetime	Until method finishes	Until GC removes it

String Concatenation:

We can concatenate strings with the plus operator ( ).

String concatenation is used to dynamically generate messages that can be different depending on context.

```
String favoriteAnimal = "parrot";

String message = "My favorite animal is the " + favoriteAnimal;

System.out.println(message);
```

o/p:

Arithmetic Operators in Java

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Examples
+	Addition	3 + 5, x + 5, 3 + y, x + y
-	Subtraction	8 - 3, 8 - y, x - 3, x - y
*	Multiplication	3 * 5, 3 * y, x * 5, x * y
/	Division	8 / 4, 8 / y, x / 8, x / y
%	Modulus	8 % 4, 8 % y, x % 4, x % y

Expression vs Statement ?

a) 2 + 3 ?
b) X = 2 ?
c) Number of expressions in (2+ 3) * 5 ?

Operator Precedence in Java:

This is the hierarchy from highest precedence to lowest precedence:

- Parentheses are always respected
- Exponentiation (raise to a power)
- Multiplication, Division and Remainder
- Addition and Subtraction
- Left to right

Ex1:

```
System.out.println(1 + 2 * 3); // Output: ?
```

```
System.out.println((1 + 2) * 3); // Output:?
```

Augmented Assignment Operators in Java

```
myVar = myVar + 5;
```

Since this is such a common pattern, there are operators which do both a mathematical operation and assignment in one step.

One such operator is the += operator:

```
int myVar = 10;
myVar += 5;
```

```
System.out.println(myVar); // Output: 15
```

Increment & Decrement Operators in Java:

In programming, increasing the value of a variable by 1 is so common, there is an operator designed to do that.

Postfix:

If used postfix, the increment operator increments and returns the value before incrementing:

```
int number = 10;
```

```
int oldNumber = number++;

System.out.println(oldNumber); // Output: 10
System.out.println(number); // Output: 11
```

Prefix:

When used prefix, the increment operator increments and returns the value after incrementing:

```
int number = 10;

int newNumber = ++number;

System.out.println(newNumber); // Output: 11
System.out.println(number); // Output: 11
```

Exercise : Buy Candy

A box of candy costs price dollars. You have balance dollars. Compute the number of boxes of candy you can buy and return how many more dollars you need to buy one more box of candy.

Example:

```
Input: price = 4, balance = 9

Output: 3
```

Exercise :Hand Shakes

There are n friends in a group. When they meet, everybody shakes hands with everybody. How many hand shakes happen?

Example:

Input: n = 3

Output: 3

Explanation:
Considering A, B, and C are the 3 friends.
The hand shakes are: (A, B), (A, C) and (B, C)

Exercise : Last Two Digit Sum

Given a non-negative integer n with at least two digits, compute and return the sum of the last two digits

Example:

Input: n = 2379

Output: 16

Control Statements in Java:

If statement:

if statements allow us to build programs that can make decisions based on some conditions. Here is the syntax:

```
boolean isRainy = true;
if(isRainy) {
    System.out.println("I bring an umbrella");
}
```

If-else Statements in Java:

Many times in our life, not only we choose to do something if a condition is met, but also choose to do something different if that condition is not met. For example:

If I'm tired:
    I take a nap
Otherwise:
    I start coding

```
bool amTired = false;
if(amTired) {
    System.out.println("I take a nap");
}
else {
    System.out.println("I start coding");
```

Comparison Operators in Java

When writing **if** statements, we usually compare two values using one or more comparison operators:

Operator	Name	Examples
==	Equal	x == 5, x == y
!=	Not equal	x != 3, x != y
>	Greater than	x > 5, x > y
<	Less than	x < 8, x < y
>=	Greater than or equal to	x >= 4, x >= y
<=	Less than or equal to	x <= 3, x <= y

```
int a = 10;
int b = 15;

if (a == b) {
    System.out.println("a is equal to b");
}
if (a < b) {
    System.out.println("a is less than b");
}
```

Exercise : Parity of Number

Given a non-negative integer n, check it's parity(Evenness) and return "even" or "odd"

A number is "even" if it is divisible by 2 and "odd" otherwise

Example:

Input: n = 12

Output: "even"

Explanation:
12 is divisible by 2, so it is even

Logical Operators: And (&&)

Sometimes you will need to test more than one thing at a time. The logical and operator (&&) returns true if and only if both conditions to the left and right of it are true. For example:

```
10 == 10 && 7 < 10 // Evaluates to true
```

Code Quality: Reducing If-Else to Boolean Expressions:

Ex:

At a contest, children can participate only if they have ages between and maxAge. Given the age of a child, check if they are allowed or not.

Example 1:

Input: minAge = 7, maxAge = 12, age = 7
Output: true

Solution:

```
System.out.println(  age > minAge && age < maxAge  );
```

Logical Operators: Or (||)

Sometimes we will need to run some code if at least one of two conditions is true. The logical or operator (||) returns true if either of the conditions is true. Otherwise, if both conditions are false, it returns false.
For example:

```
7 >= 10 || 10 < 12 // Evaluates to true
```

Logical Operators: Not (!) :

The logical not operator (!) is used to invert the value of a boolean condition. For example:

```
7 <= 10 // Evaluates to true
!(7 <= 10) // Evaluates to false

12 != 12 // Evaluates to false
!(12 != 12) // Evaluates to true

!true // Evaluates to false
```

```
bool hungry = true;
if(!hungry) {
    System.out.println("I can wait a little longer");
}
else {
    System.out.println("I need food!");
}
```

Ternary operator:

The **ternary operator** in Java (?:) is a shorthand for if-else statements. It is also known as the **conditional operator** and has the following syntax:

condition ? expression1 : expression2;

- If condition is **true**, expression1 is executed.
- If condition is **false**, expression2 is executed.

Ex:

```
int a = 10, b = 20;

int min = (a < b) ? a : b; // Assigns the smaller value to min

System.out.println("Minimum: " + min);
```

o/p ?

switch statement :

The switch statement in Java is a control flow statement used to execute one block of code from multiple possible cases based on the value of an expression.

Syntax:

```
switch (expression) {
    case value1:
        // Code to execute if expression == value1
        break;
    case value2:
        // Code to execute if expression == value2
        break;
    ...
    default:
        // Code to execute if no cases match
```

}

Key Points:

- **The expression must evaluate to byte, short, char, int, String, or enum (from Java 7+).**
- **Each case represents a possible value for the expression.**
- **break prevents fall-through to the next case.**
- **default is optional and executes if no cases match.**

Example:

```
int day = 3;

switch (day) {

    case 1:

        System.out.println("Monday");

        break;

    case 2:

        System.out.println("Tuesday");

        break;

    case 3:

        System.out.println("Wednesday");

        break;

    default:

        System.out.println("Invalid day");

}
```

Output: ?

Switch expressions

In Java 14+, enhanced switch expressions allow a more concise syntax:

```
String result = switch (day) {
    case 1 -> "Monday";
    case 2 -> "Tuesday";
    case 3 -> "Wednesday";
    default -> "Invalid day";
};
System.out.println(result);
```

This approach improves readability and eliminates the need for break statements.

Conditional Exercise Questions:

1. Find biggest among the given three values a) using nested if, b) Ladder if c) simple if
2. Find the pass grade of a student, given the marks
   >=80 → A (Very Good)
   70 - 79 → B (Good)
   60 - 69 → C (Satisfactory)
   50 - 59 → D (Pass)
   Below 50 → F (Fail)
3. A leap year happens every four years, so it's a year that is perfectly divisible by four. However, if the year is a multiple of 100 (1800, 1900, etc), the year must be divisible by 400 to be leap.
   Write a function that determines if the year is a leap year or not.

   leapYear(2020) ⟶ true

   leapYear(2021) ⟶ false

   leapYear(1968) ⟶ true

   leapYear(1900) ⟶ false

Methods in Java

A **method** in Java is a block of code that performs a specific task. It is similar to a function but is associated with a class or an object. Methods help in **code reusability, modularity, and better organization**.

Syntax:

```
returnType methodName(parameters) {
    // Method body
    return value;  // (if returnType is not void)
}
```

Note:

- **Methods can take parameters** (inputs).- as per the need - params
- **Methods may return a value** (returnType).- as per the requirement
- **Void methods** don't return anything (void).
- Indentation is not mandatory like in Python, but we use it to make our code easier to read.

Types of Methods in Java:

1. **Predefined Methods** – Built-in methods from Java libraries (e.g., Math.sqrt(25)).
2. **User-Defined Methods** – Methods created by the programmer, which can be further classified as:
    - **Static Methods** – Belong to the class and can be called without an instance (ClassName.methodName()).
    - **Instance Methods** – Belong to an object and require an instance to be invoked (obj.methodName()).

Calling a function:

The code inside a function body runs, or executes, only when the function is called.

You can call or invoke a function by typing its name followed by parentheses, like this:

```
// Function declaration:
void sayHello() {
        System.out.println("Hello World");
}

// Function call:
sayHello(); // Output: Hello World
```

All of the code inside the function body will be executed every time the function is called.

We can call a function as many times as it is needed.

Exercise: Create Function in Java

We've written this code:

```
void mainFunction() {
        System.out.println("Hello World!");
        System.out.println("Coding is amazing!");
        System.out.println("Functions are so useful!");

        System.out.println("Hello World!");
        System.out.println("Coding is amazing!");
        System.out.println("Functions are so useful!");

        System.out.println("Hello World!");
        System.out.println("Coding is amazing!");
        System.out.println("Functions are so useful!");
}
```

Write the same code in a better way by creating and calling a function `printMessages()`

Function parameters:

Function parameters allow functions to accept input(s) and perform a task using the input(s). We use parameters as placeholders for information that will be passed to the function when it is called.

When a function is defined, its parameters are specified between the parentheses that follow the function name.

Here is our function with one String parameter, audience:

```
void sayHello(String audience) {
  System.out.println("Hello " + audience);
}
```

Calling with arguments:

Then we can call sayHello() and specify the values in the parentheses that follow the function name. The values that are passed to the function when it is called are called arguments.

```
// Function declaration:
void sayHello(String audience) {
        System.out.println("Hello " + audience);
}

// Function call:
sayHello("humans"); // Output: Hello humans
```

Params vs args?

In the above example:

“Humans” ->?

The variable **audience** -> ?

what constitutes to the signature of a function?

The **signature of a function** consists of the following components:

1. **Function Name** – The identifier used to call the function.
2. **Parameter List** – The number, order, and data types of parameters.

Function Signature Does Not Include:

- Return type (in Java and most languages).
- Access modifiers (public, private, etc.).
- throws clause (exception declarations in Java).

Example in Java:

```
public int add(int a, int b) {  // Function definition
    return a + b;
}
```

Signature of this function:

```
add(int, int)
```

Only the function name and parameter types matter for identifying the function uniquely in **method overloading**.

Multiple parameters:

A function can have as many parameters as it needs. Here is a function with two parameters:

```
void sayHello(String name, int age) {
  System.out.println(name + ", aged " + age);
}
```

```
// Let's call it:
sayHello("John", 30); // Output: John, aged 30
sayHello("Mary", 26); // Output: Mary, aged 26
```

Notice that the order in which arguments are passed and assigned follows the order that the parameters are declared.

Varargs (Variable Arguments) in Java

Varargs (**variable-length arguments**) allow a method to accept **zero or more arguments** of a specified type. This helps when you don't know in advance how many arguments will be passed to a method.

Syntax:

```
returnType methodName(dataType... varName) {
    // Method body
}
```

Here, varName is the **varargs parameter**.

Example: Sum of Numbers Using Varargs

```
public class VarargsExample {
    // Method with varargs to calculate sum
    public static int sum(int... numbers) {
        int total = 0;
        for (int num : numbers) {
            total += num;
        }
        return total;
    }
    public static void main(String[] args) {
        System.out.println(sum(10, 20, 30));    // Output: 60
        System.out.println(sum(5, 15));         // Output: 20
```

```
        System.out.println(sum());          // Output: 0 (no arguments)
    }
}
```

Key Rules for Using Varargs

A method can have only one varargs parameter and it should be the last one

public void test(int... numbers, String name) { } // ❌ Invalid

✅ **Correct way:**

public void test(String name, int... numbers) { } // ✅ Valid

Varargs vs Array Parameter

A varargs parameter is internally treated as an array, but varargs allow **passing values directly** without explicitly creating an array.

Example:

public void display(int[] arr) { }   // Requires an array

public void display(int... arr) { } // Can take direct values

When to Use Varargs?

- When a method should handle **a varying number of arguments**.
- When defining **utility functions** like sum, print, format, etc.
- When **overloading multiple methods** with different argument counts.

Returning from methods in Java

In the previous example, we have used void in the function declaration. This means the function is not returning any value.

It's also possible to return a value from a function. For this, we need to specify the returnType of the function during function declaration.

Inside the function, we can use the return statement to send a value back out of a function.

The directive return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to answer below):

Variables as arguments:

Variables can also be passed as arguments for function calls:

```
void sayHello(String name, int age) {
  System.out.println(name + ", aged " + age);
}

String name = "Andy";
int age = 28;

sayHello(name, age); // Output: Andy, aged 28

sayHello("Mary", age); // Output: Mary, aged 28
```

Exercise Questions:

1. GCD :

Write a method to find the gcd of given two values

2. Fibonacci

Write a method to find the nth number in the fibonacci series

3.Factorial

Write a method to find the factorial of a given number

Recursion

Recursion is a programming technique where a method calls **itself** to solve a problem. It is useful for problems that can be **broken down into smaller subproblems** of the same type.

Ex:

```
public static int factorial(int n) {

    if (n == 0) {  // Base case

        return 1;

    }

    return n * factorial(n - 1);  // Recursive call

}
```

Key Components of Recursion

1. **Base Case** – The stopping condition that prevents infinite recursion.
2. **Recursive Case** – The part where the function calls itself with a modified parameter.

Types of Recursion

1. **Direct Recursion** – A method calls itself directly (as in the factorial example).
2. **Indirect Recursion** – A method calls another method, which in turn calls the original method.

Pros & Cons of Recursion

✅ **Pros:**

- Simplifies code for complex problems (e.g., tree traversal, backtracking).
- Reduces the need for explicit loops.

❌ **Cons:**

- Uses **more memory** (stack frames for each recursive call).
- Can cause **StackOverflowError** if the base case is missing or incorrect.

When to Use Recursion?

- When a problem can be **broken into smaller, similar subproblems**.
- When using **divide and conquer algorithms** (e.g., Merge Sort, Quick Sort).
- When working with **data structures like trees and graphs**.

Usage of Indirect Recursion

Indirect recursion is used when two or more functions call each other in a cyclic manner. It can be useful in scenarios where logic is naturally divided between multiple functions

Example of Indirect Recursion:

```
public class IndirectRecursionExample {
    public static void main(String[] args) {
        functionA(5);
    }

    static void functionA(int n) {
        if (n > 0) {
            System.out.println("A: " + n);
            functionB(n - 1);
        }
    }

    static void functionB(int n) {
        if (n > 1) {
            System.out.println("B: " + n);
            functionA(n / 2);
        }
    }
}
```

O/P:

Classes in Java

Java is an object oriented programming language. Almost everything in Java is an object, with its attributes and methods.

Class creates a user-defined data structure, which holds its own data members (attributes) and member functions (methods), which can be accessed and used by creating an instance of that class.

Classes are like a blueprint for objects outlining possible behaviors and states that every object of a certain type could have.

Classes are defined via the keyword class, like this:

```
// Define an empty class Employee
class Employee {

};
```

Objects/Instances

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. It’s not an idea anymore, it’s an actual employee, like a person named "Andrew" who is 30 years old.

You can have many employees to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

After a class is defined, we can create objects of the class using the new keyword like this:

```
// Define an empty class Employee
class Employee {

};

// inside main function:
Employee employee = new Employee();
```

Attributes/Properties

A class attribute is a Java variable that belongs to a class and is shared between all the objects of this class. Let's add some attributes to our Employee class:

```
class Employee {
        public String name;
        public int age;
};

// inside main function:
Employee emp1 = new Employee();

emp1.name = 'Andrew';
System.out.println(emp1.name); // prints 'Andrew'

Employee emp2 = new Employee();

emp2.age = 40;
System.out.println(emp2.age); // prints 40
```

We added two properties: the String `name` and the int `age`. Then, we created two different Employee objects `emp1` and `emp2`.

The attributes of objects can be accessed and modified using the dot operator (`.`), like we did above with `emp1.name` and `emp2.age`.

The keyword `public` used inside the class is called an access specifier. `name` and `age` are public attributes, whch means they can be accessed from anywhere in the code, not just inside the class code.

Constructors

In real life, we want to set the attributes specific to each object when creating it. For example, an employee named Andrew of 30 years old and another one named Mary of 25 years old. Here the constructor comes to rescue.

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

In Java the constructor method has the same name as that of its class and is always called when an object is created.

Let's add a constructor to our Employee class:

```
class Employee {
        public String name;
        public int age;

        // constructor:
        public Employee(String name, int age) {
                this.name = name;
                this.age = age;
        }
};

// inside main function:
Employee emp1 = new Employee('Andrew', 30);

System.out.println(emp1.name);

Employee emp2 = new Employee('Mary', 25);

System.out.println(emp2.age);

// prints 'Andrew' and 25 on different lines
```

The "this":

Every object in Java has access to its own address through an important refference called `this`.

The `this` reference is an implicit parameter to all member functions. Therefore, inside a member function, `this` may be used to access and manipulate the attributes of that object, like we did above with `this.name = name`.

Member functions

Member functions are functions that belong to the class:

```
class Employee {
	public String name;
	public int age;

	// constructor:
	public Employee(String name, int age) {
		this.name = name;
		this.age = age;
	}

	//member functions:
	public void printName() {
		System.out.println(name); // it is not mandatory to use "this" to access an attribute
	}

	public void printEmployee() {
		printName();
		System.out.println(age);
	}
};

// inside main function:
Employee emp1 = new Employee('Andrew', 30);

emp1.printEmployee(); // prints "Andrew", 30

Employee emp2 = new Employee('Mary', 25);

emp2.printEmployee(); // prints "Mary", 25
```

Arrays in Java

Static Arrays

An **array** in Java is a data structure that stores multiple values of the **same type** in a contiguous memory location. It is **fixed in size** once created.

A static array or fixed array is an array for which the size / length is determined when the array is created and/or allocated.

Creation:

We create a static array by using the keyword **new**, then writing the data type of the elements and the array's length inside :

```
String[] arrayOfStrings = new String[3];
```

In this program we created an array named arrayOfStrings that consists of 3 String items.

These 3 items are **null** (no value given yet) initially.

Creation & Initialization:

There is a way to initialize an array at creation by using the **=** operator with an opening bracket, end it with a closing bracket, and put a comma between each entry, like this:

```
String[] arrayOfStrings = {"Academy", "coding", "static arrays"};
```

Method 1: Declaration & Memory Allocation

```
int[] arr = new int[5];  // Creates an array of size 5
```

✔ All elements are initialized to **0** (default for int).

Method 2: Direct Initialization

```
int[] arr = {10, 20, 30, 40, 50};  // Creates an array with predefined values
```

Method 3: Multi-line Initialization

```
int[] arr;
```

//This declares an **array reference**, but it does not yet allocate memory.Its default value is **null**

```
arr = new int[]{5, 15, 25, 35};  // Correct way to initialize separately
```

✔ new int[]{...} is required when initializing separately.

Accessing static array elements in Java

Arrays are ordered, meaning each element has a numbered position known as its index. We access an array element by referring to its index number.

Arrays use zero-based indexing, so the first element in an array has an index of 0, the second element has an index of 1, etc.

We're using bracket notation **([])** with the index after the name of the array to access the element:

```
String[] animals = {"cat", "dog", "parrot"};


System.out.println(animals[0]); // Output: "cat"
System.out.println(animals[1]); // Output: "dog"
System.out.println(animals[2]); // Output: "parrot"
```

Updating static array elements in Java

Unlike strings, the entries of arrays are mutable and can be changed freely, using indices and the bracket notation:

```
int[] ourArray = {50, 40, 30};
ourArray[0] = 15;
ourArray[1] = -1;
// ourArray is now {15, -1, 30}
```

Static Array Length in Java

We can get the length of an array using the **length** property like this:

```
int[] ourArray = {50, 40, 30};

// Printing the length:

System.out.println(ourArray.length); // Output: 3
```

Exceeding static array's bounds in Java

When accessing array data with indices, the most common problem we can run into is exceeding the array's bounds.

Remember, the items of an array are indexed from 0 to length - 1. Any index which is not in this range is invalid.

When you try to access an item with an invalid index, Java raises an exception:

```
int[] ourArray = {50, 40, 30};


// Valid indices: 0, 1, 2

System.out.println(myArray[0]); // Output: 50

System.out.println(myArray[2]); // Output: 30


// Invalid indices: 3, 30, -1

System.out.println(myArray[3]); // raises IndexOutOfBounds exception

System.out.println(myArray[30]); // raises IndexOutOfBounds exception

System.out.println(myArray[-1])); // raises IndexOutOfBounds exception
```

Looping through an array using indices in Java :

In some instances, we will want to loop through an array using indices.

The indices of an array meals are all the integer values from 0 to meals.length - 1, so we can iterate over them using an index variable i:

```
String[] meals = {"pancakes", "pasta", "pizza", "avocado"};


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

```
        System.out.println("Meal number " + (i + 1) + " is " + meals[i] + ".");
}
```

The output:

Meal number 1 is pancakes.
Meal number 2 is pasta.
Meal number 3 is pizza.

Meal number 4 is avocado.

for-each:

This is very useful for iterating over sequences of elements.

A for-each loop is a special type of loop in Java which allows us to loop over the items of a collection, such as an array.

Looping through an array:

For example, an array is a sequence of items, so we can use a for loop to iterate over each item and do something with it:

```
String[] fruits = {"banana", "orange", "pear", "kivi"};


for (String fruit : fruits) {
  System.out.println("I eat " + fruit);
}
```

o/p?

Exercises :

1. Write a program to read and store n number of student marks in an integer array.
    a) Write methods to display the marks, find the min, max, avg marks.
    b) Write a method that returns new marks by increasing the marks by 15%.
    c) Write a method that finds the second max element in the array
       Ex: 34, 62, 90, 76

What is a Subarray?

A **subarray** is a contiguous portion of an array. This means that all elements in a subarray are consecutive elements from the original array.

For example, consider the array:

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

Possible subarrays include:

- `[1]`, `[2]`, `[3]`, `[4]` (single-element subarrays)
- `[1, 2]`, `[2, 3]`, `[3, 4]` (two-element subarrays)
- `[1, 2, 3]`, `[2, 3, 4]` (three-element subarrays)
- `[1, 2, 3, 4]` (full array)

Note: Subarrays must be **continuous** (i.e., `[1, 3]` is **not** a valid subarray of `[1, 2, 3, 4]`).

Kadane's Algorithm

Kadane's algorithm is an **efficient method** to find the **maximum sum subarray** in an array of integers.

Problem Statement

Given an array of integers (which may include negative numbers), find the contiguous subarray that has the **largest sum**.

Kadane's Algorithm Approach

1. **Initialize two variables:**
    - `max_sum` → Stores the maximum sum found so far.
    - `current_sum` → Keeps track of the sum of the current subarray.
2. **Iterate through the array:**
    - At each index, add the element to `current_sum`.
    - If `current_sum` becomes **less than the current element**, reset `current_sum` to the current element (i.e., start a new subarray).
    - Update `max_sum` if `current_sum` is greater than `max_sum`.
3. **Return `max_sum` as the result.**

Searching Algorithms Using Arrays

When working with arrays (without advanced data structures like hash tables or trees), the two primary searching methods are:

1. Linear Search → Works on both sorted and unsorted arrays (`O(n)`).
2. Binary Search → Works only on sorted arrays (`O(log n)`).

Linear Search (Sequential Search)

✔ Simple method that scans each element one by one.
✔ Works on both sorted and unsorted arrays.
✔ Time Complexity:

- Best Case: `O(1)` (If the element is found at the start).
- Worst Case: `O(n)` (If the element is at the end or missing).

Binary Search (Efficient for Sorted Data)

Concept

- Works only on sorted arrays.
- Uses the divide and conquer approach.
- Repeatedly divides the array into halves until the element is found.

Algorithm (Iterative Approach)

1. Set `low = 0`, `high = n-1`.
2. Find the middle element: `mid = (low + high) / 2`.
3. If `arr[mid] == key`, return `mid`.
4. If `key < arr[mid]`, search in the left half (`high = mid - 1`).
5. If `key > arr[mid]`, search in the right half (`low = mid + 1`).
6. Repeat until `low > high`.

Time Complexity

- Best Case: `O(1)` (if the middle element is the target)
- Worst Case: `O(log n)` (dividing the array in half at each step)

Binary Search (Recursive Approach) Algorithm

Algorithm:

1. Base Case:
    - If `low > high`, return `-1` (element not found).
2. Find the middle element:
    - `mid = low + (high - low) / 2` (to avoid integer overflow).
3. Check three cases:
    - If `arr[mid] == key`, return `mid` (element found).
    - If `arr[mid] < key`, recursively search in the right half (`low = mid + 1`).
    - If `arr[mid] > key`, recursively search in the left half (`high = mid - 1`).

Pseudo Code:

```
BinarySearch(arr, low, high, key):
    If low > high:
        Return -1  // Element not found
    mid = low + (high - low) / 2
    If arr[mid] == key:
        Return mid  // Element found
    Else If arr[mid] < key:
        Return BinarySearch(arr, mid + 1, high, key)  // Search right half
    Else:
        Return BinarySearch(arr, low, mid - 1, key)  // Search left half
```

2D Array in Java

A **2D array** (two-dimensional array) in Java is an array of arrays, often used to represent tables, matrices, or grids.

Declaration & Initialization

Method 1: Declare & Allocate Memory

Creating a **3×4 matrix:**

```
int[][] arr = new int[3][4];  // 3 rows, 4 columns
```

Note: all elements initialized to 0 by default.

Method 2: Declare & Initialize in One Step

Directly assigning values to a **3×3 matrix**

```
int[][] arr = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

Accessing Elements

You can access elements using **row and column indices**.

Accessing value 6 from the array:

```
System.out.println(arr[1][2]);  // Access row 1, column 2 (0-based index)
```

Iterating Over a 2D Array

Using Nested Loops

Printing the 2D array **row by row**.

```
for (int i = 0; i < arr.length; i++) {  // Row loop
    for (int j = 0; j < arr[i].length; j++) {  // Column loop
        System.out.print(arr[i][j] + " ");
    }
    System.out.println();
}
```

Using Enhanced for Loop

```
for (int[] row : arr) {
```

```
    for (int num : row) {
        System.out.print(num + " ");
    }
    System.out.println();
}
```

✔ Simplifies iteration over rows and columns.

Jagged Arrays (Variable Column Size)

Java allows **irregular** 2D arrays (rows with different column sizes).

Jagged Array in Java

A **jagged array** is a **2D array with variable column sizes**.

Unlike a normal 2D array (which has a fixed number of columns for each row), a jagged array allows each row to have a different number of columns.

Method 1: Direct Initialization

```
int[][] jagged = new int[3][];  // Declare a 2D array with 3 rows

// Assign different column sizes to each row
jagged[0] = new int[2];  // Row 0 has 2 columns
jagged[1] = new int[4];  // Row 1 has 4 columns
jagged[2] = new int[3];  // Row 2 has 3 columns

// Assign values
jagged[0][0] = 1; jagged[0][1] = 2;
jagged[1][0] = 3; jagged[1][1] = 4; jagged[1][2] = 5; jagged[1][3] = 6;
jagged[2][0] = 7; jagged[2][1] = 8; jagged[2][2] = 9;
```

Method 2: Direct Initialization

```
int[][] jagged = {
```

```
    {1, 2, 3},
    {4, 5},
    {6, 7, 8, 9}
};
```

✔ Each row can have a different number of columns.

Common Applications of 2D arrays

✅ **Matrix operations** (addition, multiplication)
✅ **Pathfinding algorithms** (e.g., maze solving)
✅ **Game boards** (e.g., Tic-Tac-Toe, Chess)

Exercises :

1. Write a program to add two 2X2 arbitrary matrices
2. Write a program to multiply two 2X2 arbitrary matrices

Dynamic Arrays:

ArrayList in Java

A big limitation of static arrays in Java is the fact that once the array is created, it has a fixed size.

The ArrayList overcomes this limitation by having variable size and allowing for elements to be added or removed.

Creation:

We create an ArrayList by using the keywords new and ArrayList, then writing the type of the elements inside <> followed by paranthesis:

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

This code creates an empty ArrayList where each item is a String.

Casting:

A good practice is to cast the created ArrayList in its interface by using a List variable and specifying there the type of the elements, like this:

List<String> sandwich = new ArrayList<>();

Initialization:

You can also initialize an ArrayList at creation by using the List.of() function and put a comma between each entry, like this:

```
List<String> sandwich = new ArrayList<>(
    List.of("peanut butter", "jelly", "bread")
);
```

Accessing ArrayList elements in Java

We can access the data inside an ArrayList using indexes and the .get() method.

ArrayLists also use zero-based indexing, so the first element in an ArrayList has an index of 0.

```
List<Integer> array = new ArrayList<>(List.of(50, 60, 70));
System.out.println(array.get(0)); // Output: 50
int data = array.get(1);
System.out.println(data); // Output: 60
```

Updating :

The entries of ArrayLists are mutable and can be changed freely, using indices and the .set() method:

.set() takes two parameters which represent an index and a value and changes the element placed at that index with the argument value:

```
List<Integer> numbers = new ArrayList<>(List.of(50, 40, 30));
```

```
numbers.set(0, 15);
numbers.set(2, -1);

// numbers is now [15, 40, -1]
```

ArrayList Size in Java

We can get the size of an ArrayList using the .size() method:

```
List<Integer> ourArray = new ArrayList<>(List.of(50, 40, 30));

// Printing the size:

System.out.println(ourArray.size()); // Output: 3

// Changing the ArrayList:

ourArray.remove(2);

// Printing the new size:

System.out.println(ourArray.size()); // Output: 2
```

Exceeding ArrayList Bounds in Java

When accessing ArrayList data with indices, the most common problem we can run into is exceeding the ArrayList's bounds.

the items of an ArrayList are indexed from 0 to size - 1. Any index which is not in this range is invalid.

When you try to access an item with an invalid index, Java throws an error:

```
List<Integer> ourArray = new ArrayList<>(List.of(50, 40, 30));
// Valid indices: 0, 1, 2
System.out.println(myArray.get(0)); // Output: 50
System.out.println(myArray.get(2)); // Output: 30

// Invalid indices: 3, 30, -1
System.out.println(myArray.get(3)); // raises IndexOutOfBounds error
System.out.println(myArray.get(30)); // raises IndexOutOfBounds error
System.out.println(myArray.get(-1)); // raises IndexOutOfBounds error
```

Appending Item to ArrayList in Java

We can add elements to the end of an ArrayList using the .add() method.

.add() takes one parameter and "pushes" it onto the end of the ArrayList:

```
List<Integer> arr1 = new ArrayList<>(List.of(1, 2, 3));
arr1.add(4);

// arr1 is now {1, 2, 3, 4}
```

Remove Items From ArrayList in Java

Another way to change the data in an ArrayList is with the .remove() method.

.remove() takes one parameter which represents an index and removes the element placed at that index:

```
List<Integer> threeArr = new ArrayList<>(List.of(1, 4, 6));

// Remove second element:
threeArr.remove(1);

// threeArray has now the value {1, 6}

// Remove first element:
threeArr.remove(0);

// threeArray has now the value {6}
```

For Loops in Java

For loops are very useful for iterating over sequences of elements.

A for loop is a special type of loop in Java which allows us to loop over the items of a collection, such as an array.

Looping through an array:

For example, an array is a sequence of items, so we can use a for loop to iterate over each item and do something with it:

```
String[] fruits = {"banana", "orange", "pear", "kivi"};

for (String fruit : fruits) {
  System.out.println("I eat " + fruit);
}
```

Ex1: Correct the following buggy code :

So when we ran the code, we expected it to print:

I think Python is cool!
I think Java is cool!
I think JavaScript is cool!

```
String[] languages = {"Python", "Java", "JavaScript"};

for (String language : languages) {
    System.out.println("I think language is cool!");
}
```

Ex2: Correct the following buggy code :

So when we ran the code, we expected it to print:

I'm hungry!
I'll eat some pasta
I'll eat some burgers
I'll eat some pizza

```
String[] foods = {"pasta", "burgers", "pizza"};

for (String food : foods) {
    System.out.println("I'm hungry!");
```

```
    System.out.println("I'll eat some " + food);
}
```

Ex3: Correct the following buggy code :

So when we ran the code, we expected it to print:

Hey, Andy

Hey, Mike

Hey, Mary

Let's start the class!

```
String[] students = {"Andy", "Mike", "Mary"};

for (String student : students) {
    System.out.println("Hey, " + student);
    System.out.println("Let's start the class!");
}
```

Ex4: Correct the following buggy code :

So when we ran the code, we expected it to print:

What a nice day!

Hey, Andy

How are you?

Hey, Mike

How are you?

Hey, Mary

How are you?

Let's start the class!

```
String[] students = {"Andy", "Mike", "Mary"};

System.out.println("Let's start the class!");
for (String student : students) {
```

```
    System.out.println("What a nice day!");
}
System.out.println("How are you?");
System.out.println("Hey, " + student);
```

Infinite For Loops II in Java

We also use for loops to iterate over sequences like strings and arrays. We can run into problems when we manipulate a sequence while iterating on it.

For example, if we append elements to an array while iterating on it:

```
List<String> fruits = new ArrayList<>(List.of("banana", "orange"));

for(var fruit : fruits) {
  fruits.add("kivi");
}
```

Every time we enter this loop, we add a item to the end of the array that we are iterating through.

As a result, we never make it to the end of the array. It keeps growing forever!

***Note**: we want to make sure we never write infinite loops as they make our program run forever and completely unusable.*

Classical For Loop in Java:

For loops can be declared with three optional expressions separated by semicolons:

```
for (initialization; condition; iteration) {
        instruction1;
        instruction2;
        ...
```

```
}
```

Let's break down each component:

- The initialization statement is executed one time only before the loop starts and is typically used to define and set up the iterator variable.
- The condition statement is evaluated at the beginning of every loop iteration and will continue as long as it evaluates to true. When the condition is false at the start of the iteration, the loop will stop executing.
- The iteration statement is executed at the end of each loop iteration, prior to the next condition check and is usually used to update the iterator variable.

Example:

```
for (int i = 0; i < 4; i++) {
        System.out.println("Hello world!");
}

// This code prints "Hello world!" on four different lines
```

For Loops: Printing Numbers in Java

We can achieve many different outcomes with for loops.

It's all about tuning the initialization, condition and iteration statements.

For example, we can print all numbers from 2 through 6:

```
for (int i = 2; i < 7; i++) {
        System.out.println(i);
}
```

The output of this code is:

```
2
```

3

4

5

6

Break the loop in Java:

With the break statement, we can prematurely terminate a loop from inside that loop.

When Java reaches the break statement, it's going to immediately terminate the loop without checking any conditions.

Breaking a for loop:

In this example, we will terminate the loop when we find a "banana" in our array:

```
String[] fruits = {"kivi", "orange", "banana", "apple", "pear"};
for (String fruit : fruits) {
        System.out.println(fruit);
        if(fruit == "banana") {
                break;
        }
}
```

The output of this code is:

kivi

orange

banana

Continue the loop in Java:

With the continue statement we can stop the current iteration of the loop, and continue with the next.

When Java hits continue, it skips (not execute) any code left, and jumps directly to the next iteration instead.

In this example, we will not let anyone inside a bar if their age is less than 21:

```
int[] ages = {10, 30, 21, 19, 25};


for (int age : ages) {
        if (age < 21) {
                continue;
        }
        System.out.println("Someone of age " + age + " entered the bar.");
}
```

The output of this program is:

Someone of age 30 entered the bar.

Someone of age 21 entered the bar.

Someone of age 25 entered the bar.

Looping In Reverse in Java

A for loop can also count backwards, as long as we define the right conditions. For example:

```
for (int i = 7; i > 2; i--) {
        System.out.println(i);
}
```

The output of this code is:

7

6

5

4

3

Looping through an array using indices in Java :

In some instances, we will want to loop through an array using indices.

The indices of an array meals are all the integer values from 0 to meals.length - 1, so we can iterate over them using an index variable i:

```
String[] meals = {"pancakes", "pasta", "pizza", "avocado"};


for (int i = 0; i < meals.length; i++) {
        System.out.println("Meal number " + (i + 1) + " is " + meals[i] + ".");
}
```

The output of this code is:

Meal number 1 is pancakes.
Meal number 2 is pasta.
Meal number 3 is pizza.

Meal number 4 is avocado.

While loop in Java

A while loop allows your program to perform a set of instructions as long as a condition is satisfied.

Here is the structure of a while loop:

```
while(condition) {
        instruction1
        instruction2
        ...
```

}

Let's check an example:

```
int count = 1;
while(count <= 5) {
        System.out.println("Hello");
        count++;
}
System.out.println("Finished!");
```

The output of this program is:

```
Hello
Hello
Hello
Hello
Hello
Finished!
```

While Loop: Printing Numbers in Java

We can achieve many different outcomes with while loops.

Print all the numbers from 0 through 4

```
int i = 0;
while(i < 5) {
        System.out.println(i);
        i++;
}
```

Infinite while loops in Java

Loops bring great power but also great responsability. The biggest danger when using loops is writing an infinite loop.

An infinite loop is a loop that never terminates. For example, if we forget to increment i in our previous example:

```
int i = 1;
while(i < 5) {
        System.out.println(i);
        // i++; => forgot this
}
```

This program will never terminate as i will always have value 1 and thus the condition i < 5 will always be satisfied.

The program would basically print 1 forever never exiting the while loop.

Loop Exercise

Exercise 1:

Given two positive integers A and B, print to the console the first A non-negative numbers that are divisible by B (hint : multiples of B , that count to A).

A number X is divisible by B if X modulo B == 0

Example:

Input: A = 5, B = 3

Output:

3

6

9

12

15

Exercise 2: Print Positive Numbers From Array

Given an array of integers, print to the console the positive numbers, one on each line.

A number n is positive if it is strictly greater than 0.

Example:

Input: nums = [3, -2, 0, 4, -4]
Output (console):
3
4

Exercise 3: Even Numbers From Array: Buggy Code in Java

Inside the code editor we've tried to write a function that takes an array nums as argument and prints to the console the even numbers from that array.

So when we called printEvenNumbers(new int[]{2, 1, 0, 4, 3}), we expected our code to print:

2
0
4

Exercise 4: Print Even - Odd

Given a positive integer n, for each number from 0 to n - 1, print "odd" if the number is odd or the number itself if it is even.

Example:

Input: n = 6

Output (console):

0

odd

2

odd

4

Odd

Exercise 5: **Array Contains**

Given an array of integers nums and another integer value, check if value occurs in nums.

If the value occurs in nums, return true; otherwise return false.

Examples:

contains([1, 2, 4, 5], 4) -> true

contains([-1, 2, -4, 0, 10], 7) -> false

Exercise 6: **Positive Number In Array: Buggy Code in Java**

Inside the code editor we've tried to write a function that takes an array nums as argument and returns true if there exists at least one positive (greater than zero) number in the array; returns false otherwise.

So when we called the function for [-1, 2, 3], we expected our code to print:

Array has positive numbers

but it seems like we made some mistakes because when we run our code, it prints:

Array doesn't have positive numbers

```
boolean checkForPositive(int[] nums) {
    for (int num : nums) {
        if (num > 0) {
            return true;
```

```
        } else {
            return false;
        }
    }
}

// Do not change this code:
void mainFunction() {
    if (checkForPositive(new int[]{-1, 2, 3})) {
        System.out.println("Array has positive numbers");
    } else {
        System.out.println("Array doesn't have positive numbers");
    }
}
```

Correct Code : ?

—---

Exercise 6.2 : **Hurdle Jump**

Given an array of hurdle heights and a jumper's jump height, determine whether or not the hurdler can clear all the hurdles. If they can, return `true`; otherwise return `false`.

A hurdler can clear a hurdle if their jump height is greater than or equal to the hurdle height.

Examples:

`hurdleJump([1, 2, 3, 4, 5], 5) ➞ true`

`hurdleJump([5, 5, 3, 4, 5], 3) ➞ false`

`hurdleJump([5, 4, 5, 6], 10) ➞ true`

hurdleJump([1, 2, 1], 1) ⟶ false

Note: Return true for the edge case of an empty array of hurdles. (Zero hurdles means that any jump height can clear them).

Exercise 7 : **Prime Number**

Given a non-negative integer n check if it is a prime number

Prime numbers are positive numbers greater than 1 that are divisible only by 1 and the number itself.

Examples of prime numbers: 2, 3, 5, 7, 11, 13, 17, 19, 23

Example 1:

Input: n = 31
Output: true

Why Do We Check Up to sqrt(n) Only?
1. Definition of a Prime Number

- A prime number has exactly two factors: **1 and itself**.
- A composite number has **at least one additional factor**.

2. Factor Pair Property

- If n is **composite**, it can be written as the product of two factors:
  n = a * b
- If both a and b were **greater than** sqrt(n), their product would be **greater than** n, which is a contradiction.
- Therefore, at least one of a or b must be **≤ sqrt(n)**.

3. Example Illustration

- Consider n = 36. Its factors are:
  1 × 36, 2 × 18, 3 × 12, 4 × 9, 6 × 6
- Notice that **after 6 (i.e., sqrt(36)), factors just repeat in reverse**.
- So, if no factor is found **up to sqrt(n)**, then n is prime.

Exercise 8 : **Find the Sum**

1. **Sum of Squares**

Given a non-negative integer n, compute and return the sum of
$1^2 + 2^2 + 3^2 + ... + n^2$

Example:

Input: n = 3

Output: 14

Explanation:
$1^2 + 2^2 + 3^2 = 1 + 4 + 9 = 14$

2. Sum of digits

Find the Sum of digits in a given number.

Example:

Input: n = 4856

Output: 23

Explanation:
4 + 8 + 5 + 6

3. Reverse the given number:

Example:

Input: n = 356

Output: 653

Exercise 9 : Number of Occurrences in Array

Given an array of integers nums, count and return the number of occurrences of a given value.

Example:

Input: nums = [1, 4, 2, 2, 5, 2], value = 2
Output: 3
Explanation: the value 2 appears 3 times in the array

Note:

Do not use builtin functions, it would defy the purpose of the challenge. Write the whole code yourself.

Exercise 10 : **H4ck3r Sp34k**

Given a string, write a function that returns a coded (h4ck3r 5p34k) version of the string.

In order to work properly, the function should replace all "a"s with 4, "e"s with 3, "i"s with 1, "o"s with 0, and "s"s with 5.

Example 1:

Input: "programming is fun"

Output: "pr0gr4mm1ng 15 fun"

Exercise 11: **Maximum Value in Array**

Given an array of integers, return the maximum value from the array.

Example:

Input: nums = [2, 7, 11, 8, 11, 8, 3, 11]
Output: 11
Explanation: The maximum value is 11

Exercise 12: **Second maximum Value in Array**

Given an array of integers, return the maximum value from the array.

Example:

Input: nums = [2, 7, 11, 8, 11, 8, 3, 11]
Output: 8
Explanation: The second maximum value is 8

Exercise 13: **Return Odd > Even**

Given an array, return true if there are more odd numbers than even numbers, otherwise return false.

Example:

Input: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
Output: true

Explanation:
There are 5 odd numbers in the array: 1, 3, 5, 7, 9
There are 4 even numbers in the array: 2, 4, 6, 8
5 is greater than 4, so our functions should return true

Exercise 14 : **Linear Searching**

Given an array of integers nums, return the index of a given value.

If the value doesn't exist, return -1.

Example 1:

Input: nums = [1, 2, 4, 5], value = 4
Output: 2
Explanation: nums[2] is 4

Exercise 15 : **Get Full Names - 1**

Given two arrays of strings named firstNames and lastNames, return an array containing the full names (separated by one space)

Example:

Input: firstNames = ["John","Andy","Mary"]

```
lastNames = ["Doe","Smith","Johnson"]
```

Output: ["John Doe", "Andy Smith", "Mary Johnson"]

Get Full Names II

Given an array of strings `names` that contain first and last names adjecent to each other, return an array containing the full names (separated by one space)

Example:

Input: names = ["John", "Doe", "Andy", "Smith", "Mary", "Johnson"]

Output: ["John Doe", "Andy Smith", "Mary Johnson"]

Exercise 16 : **Reverse Array**

Given an array of integers nums, return an array containing the integers in reverse order, without using built-in functions.

Example:

Input: nums = [2, 9, 1, 5, 8]
Output: [8, 5, 1, 9, 2]

Note:

Do not use builtin functions such as `reverse()`, it would defy the purpose of the challenge. Write the whole code yourself.

Exercise 17 : **Print Powers**

Given two non-negative integers A and B, print to the console all numbers less than or equal to B that are powers of A

Powers of a number are: A^0, A^1, A^2, etc.

Example:

Input: A = 3, B = 100

Output:
1
3
9
27
81

Algorithm:

1) Read A
2) Read B
3) Set presentPower = 1
4) While presentPower <= B :
    print(presentPower)
    presentPower *= A

Exercise 18 : **Fizz Buzz - O(n)**

Given a positive integer n print each number from 1 to n, each on a new line.
For each multiple of 3, print "Fizz" instead of the number.
For each multiple of 5, print "Buzz" instead of the number.
For numbers which are multiples of both 3 and 5, print "FizzBuzz" instead of the number

Example:

Input: n = 15

Output:

1

2

Fizz

4

Buzz

Fizz

7

8

Fizz

Buzz

11

Fizz

13

14

FizzBuzz

Exercise 19 : .**Longest Common Prefix of Two Strings**

Given two strings s1 and s2, return their longest common prefix

Example 1:

Input: s1 = "hello", s2 = "help"

Output: "hel"

Example 2:

Input: s1 = "Andy", s2 = "Mircea"

Output: ""

Exercise 20 :**Find Largest Number**

Given a positive number S, find the largest number n such that the sum $1^2 + 2^2 + 3^2 + ... + n^2$ is less than or equal to S.

For your information, $n^2 = n * n$

Example:

Input: S = 35

Output: 4

Explanation:

$1^2 + 2^2 + 3^2 + 4^2 = 1 + 4 + 9 + 16 = 30$

and 30 <= 35

if we add 5^2 we end up with 55, which exceeds 35

therefore 4 is the answer

Exercise 21 : **Max Val and Number of Occurrences - O(n)**

Given an array of integers, return the maximum value and its number of occurences.

Example:

Input: nums = [2, 7, 11, 8, 11, 8, 3, 11]

Output: [11, 3]

Explanation: The maximum value is 11 and it appears 3 times

Exercise 22 : **Fibonacci Number**

The Fibonacci numbers, commonly denoted F(n) form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

F(0) = 0, F(1) = 1

F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate and return F(n).

Example 1:

Input: n = 2

Output: 1

Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.

Example 2:

Input: n = 3

Output: 2

Explanation: F(3) = F(2) + F(1) = 1 + 1 = 2.

Exercise 22 : **Magical Number**

A magical number is obtained from a positive number by adding its digits repeatedly until we obtain one digit.

Example 1:

Input: N = 39

Output: 3

Explanation: magicNumber(39) = magicNumber(3 + 9) = magicNumber(12) = magicNumber(1 + 2) = 3

Example 2:

Input: N = 928435

Output: 4

Explanation: 9 + 2 + 8 + 4 + 3 + 5 = 31 => 3 + 1 = 4

Strings in Java

The Java String is an object that represents a sequence of characters, each identified by one index.

In Java, there are multiple ways to create strings. Each approach has its own behavior in terms of memory allocation and performance.

Using String Literals

String s1 = "Hello";

String s2 = "Hello"; // Reuses the same object from the String Pool

s1==s2 // true

s1.equals(s2) // true

Behavior:

- String literals are stored in the **String Pool** inside the heap memory.
- If a string with the same value already exists in the pool, Java reuses it instead of creating a new object.
- More memory-efficient as it avoids unnecessary object creation.

Using new String()

String s3 = new String("Hello");

String s4 = new String("Hello");

s3 == s4 // false

s3.equals(s4); // true

Behavior:

- Creates a **new object in the heap**, even if an identical string exists in the String Pool.
- It does **not** reuse objects from the String Pool.
- Less memory efficient than string literals.

Using String.valueOf()

String s5 = String.valueOf(123);

Behavior:

- Converts a primitive or object to a string.
- Uses caching for character values, making it efficient.

Using String.concat()

String s6 = "Hello".concat(" World");

Behavior:

- Creates a **new string** instead of modifying the existing one.
- More efficient than using + in a loop.

Using StringBuilder

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

String s7 = sb.toString();

Behavior:

- Mutable, meaning it modifies the existing object instead of creating a new one.
- Preferred for string manipulations in loops for better performance.

Using StringBuffer (Thread-Safe)

StringBuffer sbf = new StringBuffer("Hello");

sbf.append(" World");

String s8 = sbf.toString();

Behavior:

- Similar to StringBuilder, but **thread-safe** (synchronized).
- Slightly slower than StringBuilder.

Key Differences

Method	Mutability	Memory Efficiency	Thread Safety
String Literal	Immutable	High (Uses String Pool)	Not Thread-Safe
new String()	Immutable	Low (Creates new object)	Not Thread-Safe
String.valueOf()	Immutable	High (Uses caching)	Not Thread-Safe
String.concat()	Immutable	Moderate (Creates new object)	Not Thread-Safe
StringBuilder	Mutable	High	Not Thread-Safe
StringBuffer	Mutable	High	Thread-Safe

When to Use What?

- Use **String literals** when you need fixed, reusable strings.
- Use **new String()** only when you explicitly need a new object (rarely required).
- Use **StringBuilder** when performing **multiple modifications** (efficient).
- Use **StringBuffer** if you need **thread safety** along with modifications.
- Use **String.valueOf()** when converting **primitives to strings** efficiently.

Accessing characters:

To access the characters in a string, you use the .charAt() method

The following example returns the first character of a string with the index zero:

```
String str = "Hello";

char ch = str.charAt(0);

// ch is "H"
```

Getting the length:

We can get the length of a string using the length() method.

For example, to access the last character of the string, you use the length() - 1 index:

```
String str = "Hello";

char ch = str.charAt(str.length() - 1);

// ch is "o"
```

Concatenating strings via + operator:

To concatenate two or more strings, you use the + operator:

```
String name = "John";

String str = "Hello " + name;

System.out.println(str); // prints "Hello John"
```

If you want to assemble a string piece by piece, you can use the += operator:

```
String greet = "Welcome";

String name = "John";

greet += " to the Class, ";

greet += name;

System.out.println(greet);
```

When concatenating many strings, it is recommended to use StringBuilder for performance

String slicing:

We often want to get a substring of a string. For this, we use the substring() method.

substring(startIndex, endIndex) returns the substring from startIndex to endIndex:

String str = "JavaScript";

String substr = str.substring(2, 6);

System.out.println(substr); // prints "vaSc"

The startIndex is a zero-based index at which the substring() start extraction.

The endIndex is also zero-based index before which the substring() ends the extraction. The substr will not include the character at the endIndex index.

If you omit the endIndex, the substring() extracts to the end of the string:

String str = "JavaScript";

String substr = str.substring(4);

System.out.println(substr); // "Script"

String Immutability:

In Java, String values are immutable, which means that they cannot be altered once created.

For example, the following code:

String myStr = "Bob";

myStr[0] = 'J'; // compile error!!!

cannot change the value of myStr to "Job", because the contents of myStr cannot be altered.

Note that this does not mean that myStr cannot be changed, just that the individual characters of a string literal cannot be changed. The only way to change myStr would be to assign it with a new string, like this:

```
String myStr = "Bob";

myStr = "Job";

//or

myStr = 'J' + myStr.substring(1, 3);
```

Immutability affects performance when concatenating many strings, it is recommended to use StringBuilder for performance

Convert a string to an array of characters:

When we have to change characters often in a string, we want a way of doing this quickly. Because strings are immutable, we change a character by creating a whole new string, which is O(n).

But arrays are mutable and changing an element in an array is O(1). So what we can do is first convert our string to an array of characters, operate on that array and than convert the array back to a string.

We can convert a string to an array using the toCharArray() method like this:

```
String str = "Andy";

char[] myArr = str.toCharArray();

// myArr is ['A', 'n', 'd', 'y']

myArr[0] = 'a';

myArr[2] = 'D';

// We convert an array of chars back to a string using the constructor:

str = new String(myArr);

System.out.println(str); // "anDy"
```

Iterating through the characters:

We can iterate throught the elements of a string using indices and a for loop like this:

```
String myStr = "Andy";

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

    System.out.println(myStr.charAt(i));

}
```

StringBuilder in Java :

StringBuilder is a **mutable** class in Java used for creating and manipulating strings efficiently. Unlike String (which is immutable), StringBuilder allows modifications **without creating new objects**, making it more efficient for frequent string manipulations.

Creating a StringBuilder

```
StringBuilder sb = new StringBuilder("Hello");
```

This creates a StringBuilder object with "Hello".

Important Methods:

Method	Description	Example Output
`append(str)`	Adds a string at the end	`"Hello World"`
`insert(index, str)`	Inserts at a specific index	`"Hello Java"`
`replace(start, end, str)`	Replaces part of the string	`"Hi World"`
`delete(start, end)`	Removes part of the string	`"Hello"`
`reverse()`	Reverses the string	`"olleH"`
`capacity()`	Returns buffer capacity	`16 (default)`

When to Use StringBuilder?

✔ When you need **high-performance** string modifications.
✔ When **thread safety** is **not required**.
✔ When performing **repeated string concatenation in loops**.

StringBuffer in Java

StringBuffer is a **mutable** (modifiable) sequence of characters, similar to String, but with the ability to change its contents without creating new objects. It is **thread-safe** because its methods are **synchronized**.

Key Features of StringBuffer

✔ **Mutable** – Can modify string content without creating new objects.
✔ **Thread-Safe** – Methods are synchronized, making it safe for multi-threaded environments.
✔ **Slower than StringBuilder** – Due to synchronization overhead.
✔ **Resizable** – Dynamically expands when needed.

When to Use StringBuffer?

🔹 When working with **multi-threaded applications** where multiple threads modify the same string.
🔹 When performance is not the primary concern but **thread safety is required**.
🔹 When frequent modifications to a string are needed **in a thread-safe way**.

Key Differences

Feature	`StringBuilder`	`StringBuffer`
Mutability	✅ Mutable	✅ Mutable
Thread Safety	❌ Not thread-safe	✅ Thread-safe (Synchronized methods)
Performance	🚀 Faster (No synchronization overhead)	🦕 Slower (Due to synchronization)
Use Case	Best for **single-threaded applications**	Best for **multi-threaded applications**

Exercises on Strings

Exercise 1: Reverse String

Given a string, write a function that reverses that string without using built-in functions or libraries.

Example:

`Input: "hello"`

`Output: "olleh"`

Exercise 2: Strings equality

Given two strings, check if they are equal or not without using built-in functions or libraries.

Example:

`Input:`
`"Hello"`
`“Hello”`

`Output: true`

Exercise 3: **Palindrome Check:**

Given a string, determine if it is a palindrome.

A palindrome is a word, number, phrase, or other sequence of characters which reads the same backward as forward

Example 1:

Input: "madam racecar madam"

Output: true

Example 2:

Input: "abcbx"

Output: false

Exercise 4: Reverse Words in a String - O(n)

Given an input string s, reverse the order of the words.

A word is defined as a sequence of non-space characters. The words in s will be separated by one space for simplicity.

Return a string of the words in reverse order concatenated by a single space.

Example:

Input: input = "sky is blue"

Output: "blue is sky"

Exercise 5: Sub string

You are given two strings, A and B. Your task is to determine whether the string B is present as a substring in string A. If B is a substring of A, print "YES". Otherwise, print "NO".

Example 1:

Input:
helloworld

world

Output:
Yes

Example 2:

Input:

Delhi

Hyd

Output:
No

Exercise 6:Print Triangle of Stars

Given a positive integer n print a triangle of stars as in the example below

Example:

Input: n = 5
Output:
*
**
***
****
*****
Explanation: Print 5 lines, on ith line print i stars.

Exercise 7 : Print Rhombus

Given an odd positive integer n print a rhombus of stars as in the example below

Example:

Input: n = 5

Output:

 *

 ***

*****

 ***

 *

Explanation:

There are n = 5 lines. Each line contains exactly n = 5 characters.

1st line contains two spaces, one '*' and two spaces.

2nd line contains one space, three '*' and one space.

3rd line contains five stars.

4th line contains one space, three '*' and one space.

5th line contains two spaces, one '*' and two spaces.

Exercise 8 : **Print X - $O(n_2)$**

Given a positive integer n print matrix containing an X as in the example below

Example:

Input: n = 5

Output:

x---x

-x-x-

--x--

-x-x-

x---x

Explanation:

Each line contains exactly n = 5 characters and two 'x's.

Each diagonal contains 'x'

Complexity Analysis & Big-O Notation:

Complexity analysis : is a measure of efficiency of an algorithm in terms of **time** and **space** required as the input size grows

Big O notation: to measure the performance of an algorithm.

It allows us to compare different algorithms and determine which one is better suited for large inputs.

Why Complexity Analysis?

Instead of running an algorithm on different machines and measuring execution time, which can vary due to hardware differences, we analyze how the algorithm scales with input size mathematically. This helps in choosing the best approach before implementation.

Types of Complexity

(a) Time Complexity

Time complexity refers to how the execution time of an algorithm grows as the input size increases.

(b) Space Complexity

Space complexity refers to how much extra memory an algorithm uses apart from the input data.

What is Asymptotic Analysis?

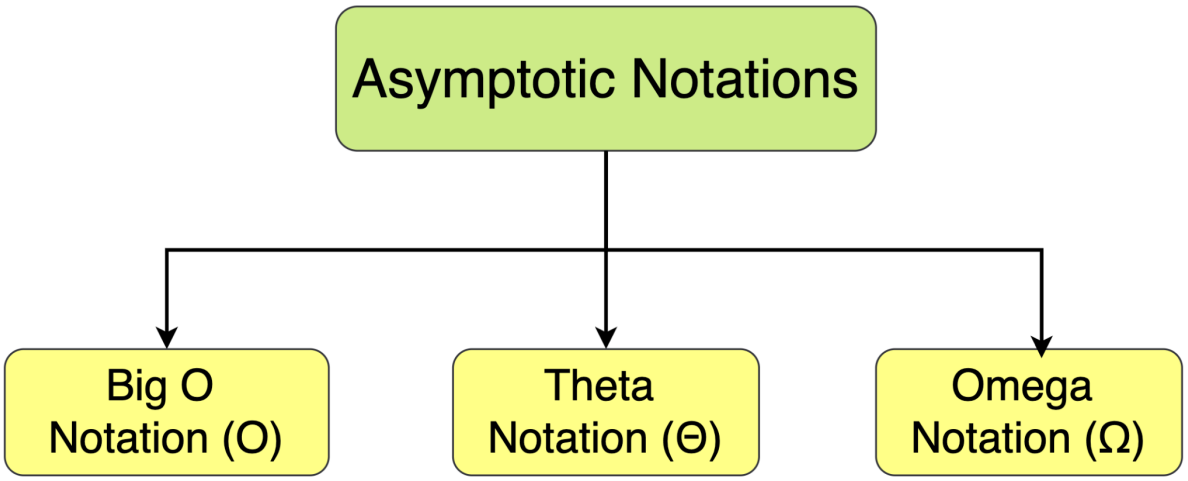
Asymptotic analysis helps us evaluate the efficiency of algorithms based on their input size.

Instead of focusing on the exact number of steps, we look at how the time or space needed by the algorithm changes as the input size (n) gets larger and larger as it reaches infinity .

Why Use Asymptotic Analysis?

1. Comparing Algorithms Easily: When you have multiple algorithms for the same problem, asymptotic analysis helps compare them by looking at their growth rates, without considering constant factors.
2. Understanding Algorithm Behavior: As the input grows, the analysis tells us which algorithms will remain efficient and which will struggle.
3. Scalability Insights: It helps identify whether an algorithm is suitable for large data sets or if it needs optimization.

Three Types of Asymptotic Notations (A Sneak Peek)



1. Big O Notation (O): Represents the upper bound of an algorithm's growth. It tells us the worst-case scenario.
2. Theta Notation (Θ): Provides a tight bound, showing both upper and lower limits on the growth rate. It describes the average-case scenario.
3. Omega Notation (Ω): Represents the lower bound of an algorithm's growth, showing the best-case scenario.

Big-O Notation

Big *O* notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity - wikipedia.

Big-O notation provides an **upper bound** on the growth rate of an algorithm's complexity.

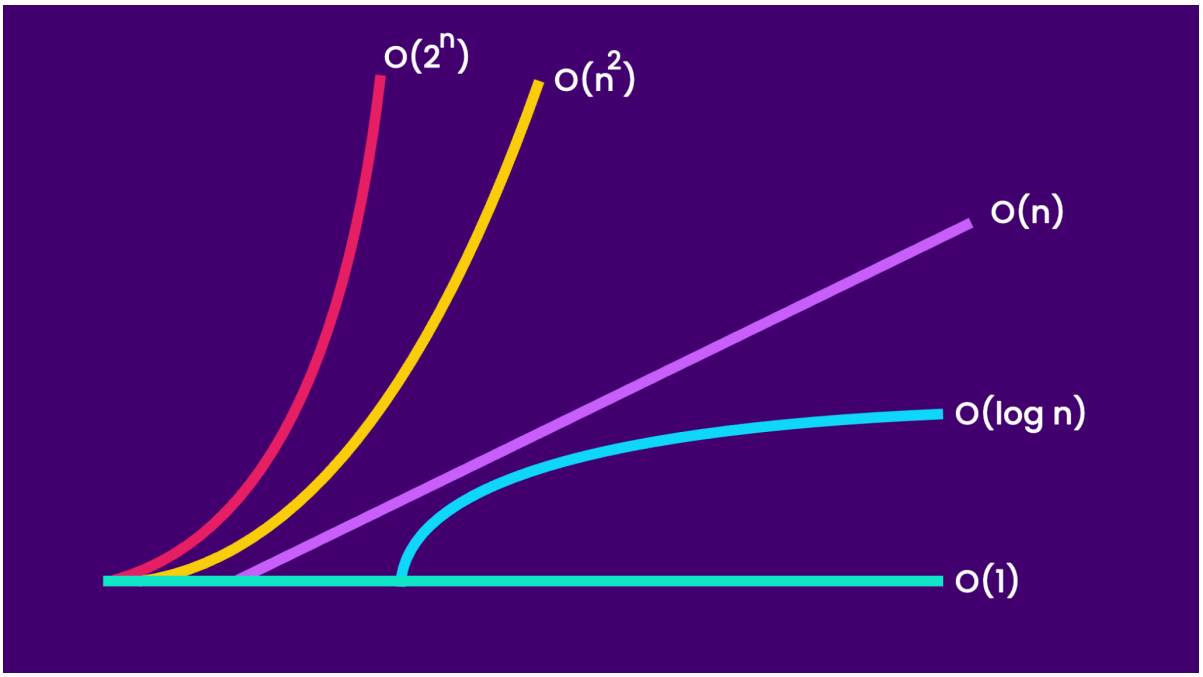
It describes the **worst-case scenario**, helping developers understand the **scalability** of their code.

We only care about how the algorithm slows down as the input grows.

Common Big-O Notations

Big-O	Name	Example
O(1)	Constant Time	Accessing an element in an array arr[i]
O(log n)	Logarithmic Time	Binary search

O(n)	Linear Time	Looping through an array
O(n log n)	Linearithmic Time	Merge sort, Quick sort (average case)
O(n^2)	Quadratic Time	Nested loops (e.g., Bubble Sort, Insertion Sort)
O(2^n)	Exponential Time	Recursive Fibonacci
O(n!)	Factorial Time	Traveling Salesman Problem (Brute force)



Examples

Example 1: O(1) – Constant Time

```
int getFirstElement(int[] arr) {
    return arr[0];  // Always takes constant time
}
```

Cost of the algorithm or runtime complexity :

Why O(1)? The execution time does not depend on n.

- This does not depend on the size of the data..
- So we call this constant time represented as O(1)

- Even if there are 2 or 100 operations, its runtime complexity is still O(1) as it does not vary with the size of an array.

Example 2: O(n) – Linear Time

```
void printArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}
```

- **Why O(n)?** The loop runs n times.

Example 3: O(n²) – Quadratic Time

```
void printPairs(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j++) {
            System.out.println(arr[i] + ", " + arr[j]);
        }
    }
}
```

- **Why O(n²)?** Nested loops make the number of operations proportional to n².

Simplifying Big-O Expressions

Rules

1. **Drop constants** → O(2n) → O(n)
2. **Drop lower order terms** → O(n + log n) → O(n)
3. **Nested loops multiply** → O(n * n) = O(n²)

What is Space Complexity?

Space Complexity measures the **amount of memory** a program uses **relative to input size (n)**. It includes:

1. **Fixed Space (O(1))** → Memory that does not change with input size.
2. **Variable Space (O(n), O(n²), etc.)** → Memory that grows with input size.

Types of Space Complexity with Examples

Constant Space – O(1)

Uses **fixed** memory, independent of input size.

Example:

```
void constantSpaceExample(int n) {
    int count = 0;  // Uses only one integer
    System.out.println(n);
}
```

Space Complexity: O(1) → The space used does **not depend** on n.

Linear Space – O(n)

Memory usage **grows linearly** with input size.

Example (Using an Array):

```
void linearSpaceExample(int n) {
    int[] arr = new int[n];  // Stores 'n' elements
    for (int i = 0; i < n; i++) {
        arr[i] = i;
    }
}
```

Space Complexity: O(n) → n elements require O(n) space.

Quadratic Space – O(n²)

Memory grows quadratically with input size.

Example (Using a 2D Array):

```
void quadraticSpaceExample(int n) {
    int[][] matrix = new int[n][n];  // Stores n × n elements
}
```

Space Complexity: O(n²) → If `n = 100`, memory grows to **10,000** elements.

Logarithmic Space – O(log n)

Used in **recursive algorithms** like **binary search**.

Example (Binary Search Recursive Call Stack):

```
int binarySearch(int[] arr, int left, int right, int key) {
    if (left > right) return -1;
    int mid = left + (right - left) / 2;
    if (arr[mid] == key) return mid;
    if (arr[mid] > key) return binarySearch(arr, left, mid - 1,
key);
    return binarySearch(arr, mid + 1, right, key);
}
```

Space Complexity: O(log n) → Each recursive call reduces `n` by half.

Exponential Space – O(2ⁿ)

Found in recursive problems like **Fibonacci without memoization**.

Example (Recursive Fibonacci):

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Space Complexity: $O(2^n)$ → Each function call branches into two new calls.

Note : According to Wikipedia, **space complexity** encompasses both the memory required to store the input (input space) and any additional memory used during computation (auxiliary space). This includes variables, data structures, and call stacks utilized by the algorithm.

However, in certain contexts, such as competitive programming or algorithm optimization, the focus is primarily on **auxiliary space**. This is because the input space is considered fixed and unavoidable, so the emphasis is on minimizing the extra memory used during computation.

Strict Definition: Space complexity **includes both input and auxiliary memory**.
Practical/Competitive Programming: People often focus **only on auxiliary space**.

Why Do Some Say "Space Complexity = Auxiliary Space"?

1. **Input storage is mandatory** – It's not considered "extra."
2. **Algorithm efficiency focuses on extra memory usage**, not the input.
3. **In competitive programming**, constraints assume input space is given, so we **optimize auxiliary space**.

Nested Loops:

A nested loop has one loop inside of another. These are typically used for working with two dimensions such as printing stars in rows and columns as shown below. When a loop is nested inside another loop, the inner loop runs many times inside the outer loop. In each iteration of the outer loop, the inner loop will be re-started. The inner loop must finish all of its iterations before the outer loop can continue to its next iteration.

Example:

```
for(int row = 0; row < 3; row++) {
        for(int col = 0; col < 5; col++) {
```

```
                System.out.print("*");
        }
    System.out.println();
}

// This code prints a rectangle with 3 rows and 5 columns, filled with '*'
```

Iterating through multidimensional arrays:

Nested loops are also helpful when we want to iterate through a multidimensional array, for example:

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

for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr[i].length; j++) {
                System.out.println(arr[i][j]);
        }
}
```

This outputs each sub-element in arr one at a time. Note that for the inner loop, we are checking the .length of arr[i], since arr[i] is itself an array.

Breaking nested loops:

In a nested loop, a break statement only stops the loop it is placed in. Therefore, if a break is placed in the inner loop, the outer loop still continues. For example:

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

for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr[i].length; j++) {
                if(arr[i][j] == 3) {
                        break;
                }
                System.out.println(arr[i][j]);
        }
}
```

```
// prints the numbers 1, 2, 5 and 6 on different lines
```

When i is 1 and j is 0, we execute the break. This means we stop the inner loop and go back to the outer loop to continue from the next iteration, which is i = 2. And as you can see, we print all the elements of row 2.

However, if the break is placed in the outer loop, all of the looping stops. For example:

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

for (int i = 0; i < arr.length; i++) {
        if(arr[i][0] == 3) {
                break;
        }
        for (int j = 0; j < arr[i].length; j++) {
                System.out.println(arr[i][j]);
        }
}
// prints the numbers 1 and 2 on different lines
```

Break with label:

A **labeled break** in Java is used to **exit a specific outer loop** from within a nested loop. Normally, break only exits the nearest enclosing loop, but with a label, we can break out of an **outer loop** directly.

Syntax of Labeled break:

```
labelName:
for (initialization; condition; update) {
    for (initialization; condition; update) {
        if (condition) {
            break labelName; // Exits the outer loop
        }
    }
```

}

- labelName is an **identifier** followed by a colon (:).
- The break labelName; statement **terminates the labeled loop** instead of just the inner loop.

Example: Exiting an Outer Loop

```
public class LabeledBreakExample {
    public static void main(String[] args) {
        outerLoop:
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                System.out.println("i = " + i + ", j = " + j);
                if (i == 2 && j == 2) {
                    break outerLoop;  // Exits the outer loop
                }
            }
        }
        System.out.println("Loop terminated.");
    }
}
```

Problem Solving Techniques & Patterns:

Problem-solving techniques help in structuring the thought process, reducing time complexity, and finding optimal solutions efficiently. Below are some of the most commonly used approaches.

These problem-solving techniques are widely used in competitive programming, interviews, and real-world applications. Mastering them will help solve complex problems efficiently.

Brute Force Approach:

The **brute force approach** is a straightforward problem-solving technique that involves **checking all possible solutions** to find the correct one. It does not use any optimization or heuristics and relies purely on **exhaustive search**.

Characteristics of Brute Force:

- **Exhaustive:** Tries every possible combination or path.
- **Simple & Naïve:** Easy to understand and implement.
- **Inefficient:** Can be slow for large inputs due to high time complexity.
- **Guaranteed to Find a Solution:** Since it checks all possibilities, it will eventually find the correct answer (if one exists).

Examples:

1. **Password Cracking:** Trying every possible combination of characters until the correct password is found.
2. **Finding a Pair in an Array:** Checking every pair of numbers to see if they add up to a target sum.
3. **String Matching:** Comparing a pattern with every possible substring in a text.
4. **Traveling Salesman Problem (TSP):** Checking all possible routes to find the shortest path.

Time Complexity:

- Often **O(n!), O(2^n), or O(n^2)** depending on the problem.

Exercises on Brute Force Approach

Exercise1 : Pair Count - O(n^2)

Given two positive integers n and sum, count the number of different pairs of integers (x, y) such that 1 <= x, y <= n and x + y equals sum

Example:

Input: n = 7, sum = 6
Output: 3
Explanation: There are 3 valid pairs: (1, 5), (2, 4), (3, 3).

Note that pairs such as (1, 5) and (5, 1) are not considered different.

Hints:

Using two nested for loops! The first loop will try out each possible x between 1 and n. The second loop will be written inside the first one and will try out each possible y between x and n.

for each x : 1 -> n:

    for each y : x -> n:

Exercise 2 : **Two Sum - O(n^2)**

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input will have at most one solution, and you may not use the same index twice.

In case no solution exists, return [-1, -1]

Example:

Input: nums = [2, 7, 11, 15], target = 9

Output: [0, 1]

Explanation: nums[0] + nums[1] = 2 + 7 = 9

Hint :

Using two nested for loops! The first loop will try out each possible i between 1 and n - 1. The second loop will be written inside the first one and will try out each possible j between i + 1 and n.

for each i : 1 -> n - 1:

    for each j : i + 1 -> n:

Importance of Writing Efficient Code in the Industry

- **Performance Optimization** – Faster execution improves user experience and system responsiveness.
- **Scalability** – Handles increasing data loads without slowing down.

- **Cost-Effectiveness** – Reduces hardware and cloud resource costs.
- **Maintainability & Readability** – Easier to debug, modify, and collaborate on.
- **Energy Efficiency** – Consumes less power, beneficial for mobile and IoT devices.
- **Competitive Advantage** – Faster applications attract and retain more users.
- **Better User Experience** – Reduces lag and improves customer satisfaction.
- **Security & Reliability** – Prevents vulnerabilities and ensures system stability.

Alternatives to Brute Force Approach

Brute force is a simple but inefficient method. Below is a list of **optimized techniques** that can be used as alternatives, depending on the problem type.

Technique	Best Use Cases	Time Complexity
Two Pointers	Sorted arrays, finding pairs/triplets, palindromes	**O(n)**
Sliding Window	Contiguous subarrays, max sum, distinct elements	**O(n)**
Hashing	Frequency count, quick lookups, duplicates	**O(1) - O(n)**
Binary Search	Searching in sorted data	**O(log n)**
Prefix Sum	Range sum queries	**O(n) (preprocess), O(1) (query)**
Backtracking	Constraint problems (N-Queens, permutations)	**O(2^n) - O(n!)**
Dynamic Programming	Overlapping subproblems (Knapsack, LCS)	**O(n) - O(n²)**
Greedy Algorithms	Optimization problems (Coin change, scheduling)	**O(n log n) - O(n)**

Two Pointer Technique

When to Use?

- Sorted arrays or linked lists
- Finding pairs, triplets, or subarrays with a given condition
- Checking for palindromes

Time Complexity: O(n) (in most cases)

Example Problem: Find two numbers in a sorted array that sum to a target
Alternative to: Nested loops (**O(n²)**)

```
// Two Pointer Approach
int left = 0, right = arr.length - 1;
while (left < right) {
    int sum = arr[left] + arr[right];
    if (sum == target) return true;
    else if (sum < target) left++;
    else right--;
}
```

Sliding Window Technique

When to Use?

- Problems involving **contiguous** subarrays or substrings
- Maximum/minimum sum, longest/shortest subarray, anagrams, distinct characters

Time Complexity: O(n) (reduces redundant calculations)

Example Problem: Find the maximum sum of k consecutive elements in an array
Alternative to: Brute-force nested loops (**O(n²)**)

```
// Sliding Window Approach
int windowSum = 0, maxSum = 0;
for (int i = 0; i < k; i++) windowSum += arr[i];

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

```
    windowSum += arr[i] - arr[i - k];  // Slide the window
    maxSum = Math.max(maxSum, windowSum);
}
```

Sorting + Binary Search

When to Use?

- Searching for elements in a sorted array
- Finding missing elements, searching range, etc.

Time Complexity: O(log n) (search), **O(n log n)** (if sorting is required)

Example Problem: Find if an element exists in an array
Alternative to: Linear search (**O(n)**)

```
// Binary Search (O(log n))
int left = 0, right = arr.length - 1;
while (left <= right) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) return true;
    else if (arr[mid] < target) left = mid + 1;
    else right = mid - 1;
}
```

Prefix Sum & Difference Array

When to Use?

- Problems that involve **range sum queries** or **frequent updates**
- Subarray sum problems

Time Complexity: O(n) (for precomputation)

Example Problem: Find the sum of elements from index l to r efficiently
Alternative to: Brute-force sum calculation (**O(n) per query**)

```
// Prefix Sum Array (Precompute in O(n))
int[] prefixSum = new int[arr.length];
prefixSum[0] = arr[0];
for (int i = 1; i < arr.length; i++)
    prefixSum[i] = prefixSum[i - 1] + arr[i];

// Query Sum in O(1)
int sum = prefixSum[r] - (l > 0 ? prefixSum[l - 1] : 0);
```

Hashing / HashMap Optimization

When to Use?

- Checking for duplicates, counting frequency, tracking visited elements
- Finding sums, anagrams, and improving search time

Time Complexity: O(1) to O(n) (depending on operations)

Example Problem: Find a pair with a given sum
Alternative to: Sorting & Two Pointers (**O(n log n)**), Brute-force (**O(n²)**)

```
// Hashing (O(n))
Map<Integer, Integer> map = new HashMap<>();
for (int num : arr) {
    if (map.containsKey(target - num)) return true;
    map.put(num, 1);
}
```

Backtracking (Optimized Brute Force)

When to Use?

- Generating permutations, combinations, and solving constraint-based problems
- Solving Sudoku, N-Queens, subset problems

Time Complexity: O(2^n) to O(n!) (can be optimized using pruning)

Example Problem: Solve N-Queens problem
Alternative to: Plain brute-force generation of all board configurations

```
// Backtracking (N-Queens Example)
void solveNQueens(int row) {
    if (row == n) { printSolution(); return; }
    for (int col = 0; col < n; col++) {
        if (isValid(row, col)) {
            board[row][col] = 'Q';
            solveNQueens(row + 1);
            board[row][col] = '.'; // Backtrack
        }
    }
}
```

Dynamic Programming (DP)

When to Use?

- Problems with **overlapping subproblems** and **optimal substructure**
- Fibonacci, knapsack, LCS, LIS

Time Complexity: O(n) to O(n²) (depends on problem)

Example Problem: Find nth Fibonacci number efficiently
Alternative to: Recursion (**O(2^n)**)

```
// DP Approach (O(n))
int[] dp = new int[n + 1];
dp[0] = 0; dp[1] = 1;
for (int i = 2; i <= n; i++)
    dp[i] = dp[i - 1] + dp[i - 2];
```

Greedy Algorithms

When to Use?

- Optimization problems where **local choices lead to global optimum**
- Activity selection, Huffman coding, Dijkstra's algorithm

Time Complexity: O(n log n) to O(n)

Example Problem: Find the minimum number of coins for a given amount
Alternative to: DP (**O(n²)**)

```
// Greedy Coin Change
Arrays.sort(coins, Collections.reverseOrder());
for (int coin : coins) {
    while (amount >= coin) {
        amount -= coin;
        count++;
    }
}
```