Design and Analysis of Algorithms (DAA)

---

1 Fibonacci Numbers with Step Count.java

📘 Concept:
This program calculates **Fibonacci numbers** and counts how many **steps (recursive calls)** are made during computation.

Algorithm Type:

- **Recursive Algorithm**
- Demonstrates **time complexity analysis**

🔍 Working:
Fibonacci series → 0, 1, 1, 2, 3, 5, 8, 13...
Defined as:

F(n)=F(n−1)+F(n−2)F(n) = F(n-1) + F(n-2)F(n)=F(n−1)+F(n−2)

⚙ Steps:

1. Base case: if n = 0 or 1 → return n
2. Recursive calls: compute F(n−1) + F(n−2)
3. Maintain a counter to count recursive calls.
4. Print both Fibonacci number and step count.

💬 Oral Questions:

- Q: What is the time complexity of Fibonacci recursion?
  A: $O(2^n)$ because each call makes two more recursive calls.
- Q: How can we optimize it?
  A: Using **Dynamic Programming** or **Memoization** to store computed results → O(n).
- Q: What does the step count show?
  A: The number of function calls → helps understand exponential growth in recursion.

---

2 Job Sequencing with Deadlines (Greedy).java

📘 Concept:
Finds the **maximum profit** by scheduling jobs before their deadlines.

Algorithm Type:

- **Greedy Algorithm**

🔍 Working:
Each job has:

- **Deadline (d)**
- **Profit (p)**

Goal: Schedule jobs to maximize profit before deadlines.

⚙️ **Steps:**

1. Sort all jobs by profit (descending order).
2. Create slots up to the maximum deadline.
3. For each job, assign it to the latest possible free slot before its deadline.
4. Calculate total profit.

💬 **Oral Questions:**

- Q: Why greedy?
  A: Because we make the locally optimal choice (highest profit first).
- Q: Time complexity?
  A: O(n log n) due to sorting.
- Q: Can two jobs have the same deadline?
  A: Yes, we assign whichever gives higher profit first.

3️⃣ FractionalKnapsackSimple.java

📘 **Concept:**
Select items to maximize profit while staying within capacity — but we can take **fractions** of items.

 Algorithm Type:

- **Greedy Approach**

⚙️ **Steps:**

1. Calculate profit/weight ratio for each item.
2. Sort by ratio (descending).
3. Pick items until the bag is full; if remaining capacity < item weight → take fraction.

💬 **Oral Questions:**

- Q: What's the difference between fractional and 0-1 knapsack?
  A: Fractional allows taking part of an item; 0-1 does not.
- Q: Time complexity?
  A: O(n log n) due to sorting.
- Q: Why greedy works here but not for 0-1 knapsack?
  A: Because in fractional, local optimum (best ratio) guarantees global optimum.

4️⃣ ZeroOneKnapsackSimple.java

📘 **Concept:**
Choose items to maximize total value without exceeding capacity — **cannot split items**.

 Algorithm Type:

- **Dynamic Programming**

⚙️ **Steps:**

1. Create a DP table dp[n+1][capacity+1].
2. If item fits, use:

   dp[i][w]=max(value[i−1]+dp[i−1][w−weight[i−1]],dp[i−1][w])dp[i][w] = max(value[i-1] + dp[i-1][w-weight[i-1]], dp[i-1][w])dp[i][w]=max(value[i−1]+dp[i−1][w−weight[i−1]],dp[i−1][w])

3. Else, carry forward previous value.
4. Final answer → dp[n][capacity].

💬 Oral Questions:

- Q: What's the time complexity?
  A: O(n * W) where W = capacity.
- Q: What's the difference from greedy?
  A: Dynamic Programming checks all possibilities; greedy fails in 0-1.
- Q: Space optimization possible?
  A: Yes — can use 1D DP array.

5⃣ NQueensSimple.java

📘 Concept:
Place N queens on an N×N chessboard so no two queens attack each other.

Algorithm Type:

- **Backtracking**

⚙ Steps:

1. Place queen column by column.
2. For each position, check if safe (no queen in same row, column, diagonal).
3. If safe → place queen and recurse for next column.
4. If no valid move → backtrack.

💬 Oral Questions:

- Q: Why backtracking?
  A: Because we try all possible configurations and backtrack when invalid.
- Q: Time complexity?
  A: O(N!).
- Q: What's a constraint satisfaction problem?
  A: Problem where we must satisfy specific conditions — N-Queens is one.

Machine Learning (ML) Projects

---

bank_ML → Bank Customer Churn Prediction (Neural Network)

📘 Concept:
Predict whether a customer will **leave the bank (churn)** using an **Artificial Neural Network (ANN)**.

🔍 **Workflow:**

1. Preprocessing (drop irrelevant columns, encode categorical data).
2. Normalize using `StandardScaler`.
3. Build ANN:
   - Input Layer → hidden layers (ReLU) → Output (Sigmoid).
4. Compile using **binary crossentropy loss**.
5. Evaluate accuracy, confusion matrix, classification report.

💬 **Oral Questions:**

- Q: Why sigmoid activation in output?
  A: Because it's a binary classification problem.
- Q: What's dropout?
  A: Technique to prevent overfitting by randomly deactivating neurons.
- Q: What metric used?
  A: Accuracy, precision, recall, confusion matrix.

---

diabetes_ML → Diabetes Prediction using KNN

📘 **Concept:**
Predict if a person has diabetes based on medical attributes.

□ **Algorithm:**

- **K-Nearest Neighbors (KNN)** — classification based on neighbors.

⚙□ **Steps:**

1. Normalize data (`StandardScaler`).
2. Choose `k` (e.g., 7).
3. For each test sample → find k nearest neighbors → majority vote.
4. Compute confusion matrix, accuracy, precision, recall.

💬 **Oral Questions:**

- Q: Why normalization important?
  A: Because KNN is distance-based.
- Q: What's the distance metric?
  A: Euclidean distance.
- Q: How to find best k?
  A: Elbow method (plot error rate vs k).

---

email → Email Spam Classification (KNN & SVM)

📘 **Concept:**
Classify emails as **Spam (1)** or **Ham (0)** using text-based features.

□ **Algorithms:**

- **KNN** (distance-based)
- **SVM** (margin-based linear classifier)

⚙️ **Steps:**

1. Text preprocessing → TF-IDF vectorization.
2. Split train/test.
3. Train both models.
4. Evaluate and compare accuracy.

💬 **Oral Questions:**

- Q: What's TF-IDF?
  A: Converts text to numerical vectors by weighting words based on frequency and importance.
- Q: Why SVM better?
  A: It separates classes with maximum margin and handles high-dimensional text data better.
- Q: Kernel used?
  A: Linear kernel (good for text data).

---

sales_ML_ELBOW → Sales Data Clustering (K-Means + Hierarchical)

📘 **Concept:**
Group similar sales data points into clusters.

 Algorithms:

- **K-Means Clustering** (Partition-based)
- **Hierarchical Clustering**

⚙️ **Steps:**

1. Select numeric data.
2. Standardize using `StandardScaler`.
3. Apply **Elbow Method** to find optimal clusters (k).
4. Perform clustering and visualize.
5. Use dendrogram for hierarchical clustering.

💬 **Oral Questions:**

- Q: What's inertia?
  A: Sum of squared distances of samples to their cluster center.
- Q: Difference between K-Means and Hierarchical?
  A: K-Means is partition-based; Hierarchical builds a tree structure.
- Q: Why scale features?
  A: To ensure all features contribute equally.

---

uber → Uber Fare Prediction (Regression)

📘 **Concept:**
Predict Uber ride fare using linear and ensemble regression models.

□ **Algorithms:**

- Linear Regression
- Random Forest Regressor

⚙□ **Steps:**

1. Data cleaning, remove outliers.
2. Feature selection (longitude, latitude, passenger_count).
3. Train-test split.
4. Fit models, evaluate with R^2 and RMSE.

💬 **Oral Questions:**

- Q: What's the difference between Linear Regression and Random Forest?
  A: Linear assumes linear relation; Random Forest handles nonlinearities.
- Q: What does R^2 mean?
  A: Fraction of variance explained by model (closer to 1 is better).
- Q: Why remove outliers?
  A: They distort regression line and reduce accuracy.

□💻 Blockchain Technology (BT)

BT_Bank.sol

📘 **Concept:**
A **Solidity smart contract** simulating a simple **banking system**.

□ **Concepts Used:**

- Blockchain
- Smart Contracts
- Ether transfer
- State variables & functions

⚙□ **Likely Features:**

- Deposit & Withdraw
- Account balance tracking
- Events for logging transactions

💬 **Oral Questions:**

- Q: What's a smart contract?
  A: A self-executing contract with rules directly written in code.
- Q: What's gas in Ethereum?
  A: Fee required to execute transactions.
- Q: What data type is used for Ether amounts?
  A: `uint` (unsigned integer).

BT_Student.sol

📘 Concept:
Smart contract for managing **student records** on blockchain.

⚙️ Features:

- Add or update student info
- Map student ID to details
- Retrieve data transparently

💬 Oral Questions:

- Q: What's `mapping` in Solidity?
  A: Key-value pair used for storage.
- Q: Can we change data once written?
  A: Only through functions — immutable otherwise.
- Q: What's the advantage over traditional DB?
  A: Transparency, security, decentralization.

README.md

Contains overview, instructions to run all files, datasets used, and results summaries.

✅ Summary for Oral Preparation

Category	Files	Core Concept
DAA	Fibonacci, Job Sequencing, Fractional/0-1 Knapsack, NQueens	Recursion, Greedy, DP, Backtracking
ML	bank_ML, diabetes_ML, email, sales_ML_ELBOW, uber	Classification, Clustering, Regression, Neural Networks
BT	BT_Bank.sol, BT_Student.sol	Smart Contracts, Blockchain, Solidity

1.

```
import java.util.*;


public class Fibonacci {
    // Recursive Fibonacci
    static int fibRec(int n) {
        if (n <= 1)
            return n;
        return fibRec(n - 1) + fibRec(n - 2);
    }

    // Non-recursive Fibonacci
    static void fibNonRec(int n) {
        int a = 0, b = 1;
        System.out.print(a + " " + b + " ");
        for (int i = 2; i < n; i++) {
            int c = a + b;
            System.out.print(c + " ");
            a = b;
            b = c;
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter n: ");
        int n = sc.nextInt();

        System.out.println("Non-Recursive Fibonacci:");
        fibNonRec(n);
        System.out.println("\nRecursive Fibonacci (last term): " + fibRec(n - 1));
    }
}
//Recursive:
//□ Time: O(2ⁿ)
//💾 Space: O(n) (stack calls)

//Non-Recursive:
//□ Time: O(n)
//💾 Space: O(1
```

Perfect — this is a **Java program** that demonstrates **two ways to generate Fibonacci numbers**:

1️⃣ **Recursive method** → uses function calls that call themselves.
2️⃣ **Non-recursive (iterative) method** → uses loops and variables to compute results step by step.

Let's go **line-by-line** with **concepts explained** 👇

🧩 Code Breakdown and Explanation

```
import java.util.*;
```

- Imports all classes from the `java.util` package (like `Scanner`, `ArrayList`, etc.).
- Here, we only use **`Scanner`** for input.

```
public class Fibonacci {
```

- Defines a **public class** named `Fibonacci`.
- In Java, every program must have a class, and the file name must match it (`Fibonacci.java`).

🌀 Recursive Fibonacci Function

```
static int fibRec(int n) {
```

- Declares a **static** method `fibRec` returning an integer.
- `static` means it can be called without creating an object (directly via the class).

```
    if (n <= 1)
        return n;
```

- **Base case** of recursion:
  - If `n` is 0 or 1, the Fibonacci number is the same as `n`.
  - Prevents infinite recursion.

```
    return fibRec(n - 1) + fibRec(n - 2);
```

- **Recursive case**:
  - Each Fibonacci number = sum of the previous two numbers.
  - Calls the same function twice with smaller arguments.

🧠 Concept: Recursion

- A function that **calls itself** until it reaches a base case.
- Each recursive call adds a new frame to the **call stack**.

🧮 Example for `n=5`

```
fibRec(5)
= fibRec(4) + fibRec(3)
= (fibRec(3) + fibRec(2)) + (fibRec(2) + fibRec(1))
...
```

⚙️ Non-Recursive (Iterative) Fibonacci Function

```
static void fibNonRec(int n) {
```

- A **static** method returning nothing (`void`).
- It prints Fibonacci series up to `n` terms using iteration.

```
int a = 0, b = 1;
```

- Initializes the **first two Fibonacci numbers**:
  - a = F(0)
  - b = F(1)

```
System.out.print(a + " " + b + " ");
```

- Prints the first two terms directly.

```
for (int i = 2; i < n; i++) {
```

- Starts from index 2 since first two numbers are already printed.
- The loop continues until it reaches the nth term.

```
    int c = a + b;
```

- Computes the next Fibonacci number by adding the previous two.

```
    System.out.print(c + " ");
```

- Prints the next Fibonacci number.

```
    a = b;
    b = c;
```

- **Updates** a and b for the next iteration.
  - a takes the value of b,
  - b takes the value of c.

□ **Concept: Iteration**

- Solving a problem using **loops (for/while)** instead of recursive calls.
- It's more efficient in both **time** and **space**.

🚀 Main Method

```
public static void main(String[] args) {
```

- **Entry point** of the Java program.
- The JVM starts execution from here.

```
Scanner sc = new Scanner(System.in);
System.out.print("Enter n: ");
int n = sc.nextInt();
```

- Creates a **Scanner object** to read input from the user.
- Reads an integer n → number of Fibonacci terms to generate.

□ **Concept: Scanner**

- A class in Java for reading input (from keyboard, files, etc.).
- nextInt() reads an integer value.

```
System.out.println("Non-Recursive Fibonacci:");
fibNonRec(n);
```

- Calls the **non-recursive** Fibonacci method and prints the sequence.

```
System.out.println("\nRecursive Fibonacci (last term): " + fibRec(n - 1));
```

- Calls the **recursive** method to compute the **last term only** (n-1 because indexing starts at 0).
- Prints the last Fibonacci number computed recursively.

□ Concept Summary

Concept	Explanation
Recursion	A function calling itself to solve smaller subproblems. Used in divide-and-conquer algorithms.
Iteration	Repeating a set of statements until a condition is false. Uses loops, no call stack.
Base Case	Condition that stops recursion and prevents infinite calls.
Call Stack	Memory structure that stores active method calls. Grows with recursion depth.
Time Complexity	Recursive: $O(2^n)$ (exponential), Iterative: $O(n)$ (linear).
Space Complexity	Recursive: $O(n)$ (stack), Iterative: $O(1)$ (constant).

□ Sample Output

```
Enter n: 7
Non-Recursive Fibonacci:
0 1 1 2 3 5 8
Recursive Fibonacci (last term): 8
```

⚡ Improvements / Best Practices

- For large n, recursion is **inefficient** — use **Dynamic Programming (DP)** or **iteration**.
- Always **close the Scanner**:
- sc.close();
- You could store Fibonacci numbers in an array or list for later use.

```
import java.util.PriorityQueue;

// Node class for Huffman Tree
class Node implements Comparable<Node> {
    char ch;
    int freq;
    Node left, right;

    Node(char ch, int freq) {
        this.ch = ch;
        this.freq = freq;
        this.left = null;
        this.right = null;
    }

    // Compare nodes by frequency (for greedy selection)
    public int compareTo(Node n) {
        return this.freq - n.freq;
    }
}
public class HuffmanEncodingSimple {
    // Function to print Huffman Codes
    static void printCodes(Node root, String code) {
        if (root == null) return;
        // If leaf node, print its code
        if (root.left == null && root.right == null)
            System.out.println(root.ch + " : " + code);

        // Traverse left and right subtrees
        printCodes(root.left, code + "0");
        printCodes(root.right, code + "1");
    }

    public static void main(String[] args) {
        // Step 1: Characters and their frequencies
        char[] chars = {'A', 'B', 'C', 'D', 'E'};
        int[] freq = {5, 9, 12, 13, 16};
        // Step 2: Create a min-heap (PriorityQueue)
        PriorityQueue<Node> pq = new PriorityQueue<>();

        // Step 3: Add all characters to queue
        for (int i = 0; i < chars.length; i++)
            pq.add(new Node(chars[i], freq[i]));

        // Step 4: Build Huffman Tree using greedy method
        while (pq.size() > 1) {
            Node left = pq.poll();
            Node right = pq.poll();

            // Create new internal node with combined frequency
            Node newNode = new Node('-', left.freq + right.freq);
            newNode.left = left;
            newNode.right = right;

            pq.add(newNode);
        }

        // Step 5: Print Huffman Codes
        System.out.println("Huffman Codes are:");
        printCodes(pq.peek(), "");
    }
}
```

□ Code Explanation (Line by Line)

✅ Import Statement

```
import java.util.PriorityQueue;
```

- Imports Java's **PriorityQueue** class — a **min-heap** data structure.
- It automatically keeps the smallest element (based on `compareTo`) at the top.
- Used to select the two least frequent nodes efficiently in each iteration.

□ Node Class for Huffman Tree

```
class Node implements Comparable<Node> {
```

- Defines a **Node class** to represent each element (leaf/internal node) of the **Huffman tree**.
- Implements `Comparable<Node>` so that `PriorityQueue` can order nodes based on frequency.

```
char ch;
int freq;
Node left, right;
```

- ch: the **character**.
- freq: frequency (how often it appears in the input).
- left, right: pointers to the left and right child nodes in the Huffman tree.

```
Node(char ch, int freq) {
    this.ch = ch;
    this.freq = freq;
    this.left = null;
    this.right = null;
}
```

- Constructor initializes a node with the given character and frequency.
- By default, both child pointers are null.

```
public int compareTo(Node n) {
    return this.freq - n.freq;
}
```

- Overrides compareTo() to compare nodes by frequency.
- This ensures the PriorityQueue behaves as a **min-heap** (lowest frequency node first).
- **Concept:** Comparison-based ordering → used by heaps, trees, sorting algorithms.

🌳 Main HuffmanEncodingSimple Class

```
public class HuffmanEncodingSimple {
```

- Main class that implements Huffman Encoding using the Node structure.

□ Recursive Function to Print Huffman Codes

```
static void printCodes(Node root, String code) {
```

- Recursively traverses the Huffman tree to generate binary codes for each character.
- code accumulates “0” for left and “1” for right traversal.

```
if (root == null) return;
```

- Base condition: stops recursion when there’s no node.

```
if (root.left == null && root.right == null)
    System.out.println(root.ch + " : " + code);
```

- When a **leaf node** is reached (character node, not internal), prints its character and corresponding Huffman code.
- Leaf nodes represent actual characters; internal nodes represent combined frequencies.

```
    printCodes(root.left, code + "0");
    printCodes(root.right, code + "1");
```

- Recursive traversal:
  - Move **left** → append `"0"` to code.
  - Move **right** → append `"1"`.
- Uses **pre-order traversal** pattern.

□ **Concept:**

- **Tree Traversal** — visiting nodes in a particular order (preorder, inorder, postorder).
- Here, recursion is used for tree traversal.

□ Main Method (Algorithm Execution)

```
public static void main(String[] args) {
```

- Entry point of the program.

Step 1: Input Characters and Frequencies

```
char[] chars = {'A', 'B', 'C', 'D', 'E'};
int[] freq = {5, 9, 12, 13, 16};
```

- Defines 5 characters with their frequencies (how often each occurs in data).
- Example: In real text, ‘E’ is more frequent than ‘Z’, so it should get a shorter code.

Step 2: Create a Priority Queue

```
PriorityQueue<Node> pq = new PriorityQueue<>();
```

- Creates a **min-heap** (automatically ordered by node frequency).
- This helps efficiently select two nodes with the smallest frequencies — a **greedy step**.

Step 3: Add Characters to Queue

```
for (int i = 0; i < chars.length; i++)
    pq.add(new Node(chars[i], freq[i]));
```

- Wraps each character and frequency into a `Node` and adds it to the heap.

□ **Concept:**

- Each character starts as an individual tree (single node).
- We will repeatedly merge the two smallest trees to form a bigger one.

Step 4: Build Huffman Tree

```
while (pq.size() > 1) {
```

- Continue until only one tree remains — the complete Huffman tree.

```
Node left = pq.poll();
Node right = pq.poll();
```

- Removes (polls) two nodes with the **lowest frequencies**.
- poll() removes and returns the root of the min-heap (the smallest element).

```
Node newNode = new Node('-', left.freq + right.freq);
newNode.left = left;
newNode.right = right;
```

- Creates a new **internal node**:
  - Character is '-' (non-leaf node placeholder).
  - Frequency is the **sum** of the two smallest nodes.
  - Left and right children link the two nodes being merged.

```
pq.add(newNode);
```

- Adds the newly combined node back into the priority queue.
- The process continues, merging smallest pairs until one node remains.

☐ **Concept:**

- **Greedy Algorithm** — always combine the two smallest frequency nodes at each step to achieve an **optimal prefix-free code**.

Step 5: Print Huffman Codes

```
System.out.println("Huffman Codes are:");
printCodes(pq.peek(), "");
```

- pq.peek() returns the root of the final Huffman tree (the only remaining node).
- Calls the recursive printCodes() to display binary codes for all characters.

☐ Example Output

```
Huffman Codes are:
E : 0
D : 10
C : 110
A : 1110
B : 1111
```

(Note: The exact code pattern may vary depending on tree structure but will maintain prefix-free property.)

□ Conceptual Deep Dive: Huffman Encoding

Concept	Explanation
Goal	Compress data by assigning **shorter binary codes** to more frequent characters.
Type	Greedy Algorithm — builds solution step by step choosing the local optimum (smallest frequencies).
Data Structure Used	Binary Tree + Priority Queue (Min-Heap).
Prefix Property	No code is a prefix of another (ensures unambiguous decoding).
Traversal Method	Preorder traversal (recursively builds binary codes).

□ Complexity Analysis

Step	Time Complexity	Space Complexity
Building heap	O(n)	
Extracting + merging nodes	O(n log n)	
Tree traversal (printing codes)	O(n)	
Overall	**O(n log n)**	**O(n)**

□ Connections to Subject Areas

Subject	Concept Used
Data Structures	Binary Tree, Priority Queue (Heap), Recursion
Algorithm Design	Greedy Method, Optimal Merge Pattern
OOP (Java)	Classes, Objects, Constructor, Interfaces (Comparable), Encapsulation
Complexity Analysis	Big O Notation for efficiency evaluation

⚡ Improvements and Extensions

- Accept **dynamic input** for characters and frequencies.
- Implement **decoding function** (convert binary back to text).
- Handle **edge cases** (single character input, invalid frequencies).
- Visualize tree using a traversal diagram (optional GUI/graph library).

```java
import java.util.*;

class Item {
    int weight;
    int value;

    Item(int weight, int value) {
        this.weight = weight;
        this.value = value;
    }
}

public class FractionalKnapsackSimple {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Step 1: Input items
        System.out.print("Enter number of items: ");
        int n = sc.nextInt();

        Item[] items = new Item[n];
        for (int i = 0; i < n; i++) {
            System.out.print("Enter value and weight of item " + (i + 1) + ": ");
            int value = sc.nextInt();
            int weight = sc.nextInt();
            items[i] = new Item(weight, value);
        }

        // Step 2: Input Knapsack capacity
        System.out.print("Enter Knapsack capacity: ");
        int capacity = sc.nextInt();
        // Step 3: Sort items by value/weight ratio (greedy choice)
        Arrays.sort(items, (a, b) -> Double.compare((double) b.value / b.weight, (double) a.value / a.weight));
        // Step 4: Take items into knapsack
        double totalValue = 0.0;
        int remaining = capacity;

        for (Item item : items) {
            if (item.weight <= remaining) {
                totalValue += item.value;
                remaining -= item.weight;
            } else {
                // Take fraction of the remaining weight
                totalValue += item.value * ((double) remaining / item.weight);
                break;
            }
        }
        // Step 5: Print result
        System.out.println("\nMaximum value in Knapsack = " + totalValue);
    }
}
```

□ Code Walkthrough — Line by Line

🎁 Item Class Definition

```
class Item {
    int weight;
    int value;
```

- Defines a **blueprint** (class) for each item.
- Each item has two properties:
    - `weight`: how heavy the item is.
    - `value`: how valuable the item is.

□ Concepts used:

- **OOP (Object-Oriented Programming)** — data is encapsulated in objects.
- **Encapsulation** — keeps related data (value, weight) bundled together.

```
    Item(int weight, int value) {
        this.weight = weight;
        this.value = value;
    }
```

- **Constructor** to initialize an `Item` object with given weight and value.
- `this` keyword differentiates between instance variables and constructor parameters.

□ Main Class

```
public class FractionalKnapsackSimple {
```

- Defines the main class containing the algorithm.

□ Main Method – Execution Starts Here

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
```

- Entry point of the program.
- Creates a `Scanner` object for user input.

□ Step 1: Input Items

```
System.out.print("Enter number of items: ");
int n = sc.nextInt();
```

- Takes the number of items from the user.

```
Item[] items = new Item[n];
```

- Creates an array of `Item` objects.

```
for (int i = 0; i < n; i++) {
    System.out.print("Enter value and weight of item " + (i + 1) + ": ");
    int value = sc.nextInt();
    int weight = sc.nextInt();
    items[i] = new Item(weight, value);
}
```

- Iterates through each item:
  - Reads its value and weight.
  - Creates an `Item` object and stores it in the array.
- Note: **order of arguments** — `new Item(weight, value)` matches the constructor parameters.

□ Concept: Array of Objects

- Each array element holds a reference to an `Item` object.

🎒 Step 2: Input Knapsack Capacity

```
System.out.print("Enter Knapsack capacity: ");
int capacity = sc.nextInt();
```

- The **maximum weight** that the knapsack can carry.

⚙️ Step 3: Sort Items by Value-to-Weight Ratio

```
Arrays.sort(items, (a, b) -> Double.compare((double) b.value / b.weight, (double) a.value / a.weight));
```

- Sorts items in **descending order** of value/weight ratio.
- This implements the **greedy strategy** — always choose the most valuable item per unit weight first.

Concept: Greedy Choice Property

- Always pick the locally optimal solution (highest value per weight) hoping it leads to a globally optimal solution.

💡 Why Double.compare?

- It handles floating-point comparisons safely (avoids precision errors).

💰 Step 4: Select Items for Knapsack

```
double totalValue = 0.0;
int remaining = capacity;
```

- totalValue: keeps track of total profit/value collected.
- remaining: remaining weight capacity in the knapsack.

```
for (Item item : items) {
```

- Enhanced **for-each loop** to iterate through sorted items.

✅Case 1: Item Fits Entirely

```
if (item.weight <= remaining) {
    totalValue += item.value;
    remaining -= item.weight;
}
```

- If the item's weight ≤ remaining capacity:
  - Take the **whole item**.
  - Add its value to total value.
  - Subtract its weight from remaining capacity.

Case 2: Item Doesn't Fit Entirely

```
else {
    totalValue += item.value * ((double) remaining / item.weight);
    break;
}
```

- If the item is too heavy:
  - Take **fraction** of the item that fits.
  - Value added = (fraction of weight taken × item's value).
  - break because the knapsack is now full.

□ Concept: Fractional Knapsack

- You are **allowed to take parts of an item** (e.g., half an item).
- Different from **0/1 Knapsack**, where you either take the whole item or none.

🖨□ Step 5: Display Result

```
System.out.println("\nMaximum value in Knapsack = " + totalValue);
```

- Prints the maximum achievable value within the given capacity.

📊 Example Input/Output

Input:

```
Enter number of items: 3
Enter value and weight of item 1: 60 10
Enter value and weight of item 2: 100 20
Enter value and weight of item 3: 120 30
Enter Knapsack capacity: 50
```

Output:

```
Maximum value in Knapsack = 240.0
```

□ Explanation:

- Ratios:
  - Item1 = 6
  - Item2 = 5
  - Item3 = 4
- Pick items 1 & 2 completely → 30 weight used, 160 value gained.
- Take 20/30 of item 3 → (20/30 × 120 = 80).
- Total = 240 value for 50 weight.

□ Conceptual Deep Dive

Concept	Explanation
Knapsack Problem	Classic optimization problem — maximize profit within a limited weight capacity.
Fractional Knapsack	Items can be divided (take fraction). Solved using Greedy Algorithm.
Greedy Algorithm	At each step, choose the best immediate (local) option.
0/1 Knapsack	Dynamic Programming problem (items cannot be divided).
Value/Weight Ratio	Determines which item gives the most value per unit weight.

☐ Complexity Analysis

Step	Time Complexity	Space Complexity
Sorting (greedy step)	O(n log n)	O(1)
Iteration (selection)	O(n)	O(1)
Overall	**O(n log n)**	**O(1)**

✅ **Efficient** for fractional case — no need for DP tables.

☐ Connections to Subject Areas

Subject	Concept Used
Algorithm Design	Greedy strategy, optimal substructure
Data Structures	Arrays, sorting using comparator
OOP (Java)	Classes, constructors, object arrays
Mathematics / Optimization	Ratio comparison, proportional division
Complexity Analysis	Time & space complexity, Big O notation

⚡ Best Practices and Extensions

- ✅ **Close the Scanner**:
- sc.close();
- ✅ Use List<Item> instead of arrays for flexibility.
- ✅ Format output using System.out.printf("%.2f", totalValue); for cleaner decimal display.
- ✅ Extend to handle dynamic user input, validation, or file-based data.

```
public class ZeroOneKnapsackSimple {

    // Function to solve 0-1 Knapsack using Dynamic Programming
    static int knapSack(int capacity, int weight[], int value[], int n) {
        int dp[][] = new int[n + 1][capacity + 1];

        // Build table dp[][] in bottom-up manner
        for (int i = 0; i <= n; i++) {
            for (int w = 0; w <= capacity; w++) {
                if (i == 0 || w == 0)
                    dp[i][w] = 0;
                else if (weight[i - 1] <= w)
                    dp[i][w] = Math.max(value[i - 1] + dp[i - 1][w - weight[i - 1]], dp[i - 1][w]);
                else
                    dp[i][w] = dp[i - 1][w];
            }
        }
        return dp[n][capacity];
    }

    public static void main(String[] args) {
        int value[] = {60, 100, 120};
        int weight[] = {10, 20, 30};
        int capacity = 50;
        int n = value.length;

        System.out.println("Maximum value in Knapsack = " + knapSack(capacity, weight, value, n));
    }
}
```

□ Code Explanation (Line by Line)

📘 Class Declaration

```
public class ZeroOneKnapsackSimple {
```

- Defines a public Java class named `ZeroOneKnapsackSimple`.
- Contains both the DP function and the main driver method.

⚙️ Knapsack Function Declaration

```
static int knapSack(int capacity, int weight[], int value[], int n) {
```

- A **static** method (no object creation needed) that returns an integer.
- Parameters:
    - `capacity`: total weight capacity of the knapsack.
    - `weight[]`: array containing weights of all items.
    - `value[]`: array containing values of all items.
    - `n`: total number of items.

Concept:
This function uses **Dynamic Programming (Bottom-Up Approach)** to compute the optimal result efficiently.

Step 1: Create DP Table

```
int dp[][] = new int[n + 1][capacity + 1];
```

- `dp[i][w]` will represent **the maximum value** that can be obtained using **first `i` items** and **capacity `w`**.
- Dimensions are `(n+1) × (capacity+1)` because we include the **0th row and column** for base conditions (empty set and zero capacity).

🔁 Step 2: Build DP Table (Bottom-Up)

```
for (int i = 0; i <= n; i++) {
    for (int w = 0; w <= capacity; w++) {
```

- Two nested loops:
    - `i`: number of items considered.
    - `w`: current capacity limit.

⚖️ Step 3: Handle Base Conditions

```
if (i == 0 || w == 0)
    dp[i][w] = 0;
```

- If there are **no items (`i == 0`)** or **capacity is 0 (`w == 0`)**,
  → maximum value is 0 (nothing can be added).

🎁 Step 4: Include or Exclude Current Item

```
else if (weight[i - 1] <= w)
    dp[i][w] = Math.max(value[i - 1] + dp[i - 1][w - weight[i - 1]], dp[i - 1][w]);
```

- **Check if the current item can fit in the current capacity `w`.**
- If yes, two choices:

    1. **Include the item:**
       Add its value and reduce capacity by its weight →
       `value[i - 1] + dp[i - 1][w - weight[i - 1]]`
    2. **Exclude the item:**
       Don't include it →
       `dp[i - 1][w]`
- Choose the **maximum** of the two options.

□ **Concept:**
This is the **optimal substructure property** —
The solution to a problem depends on the solutions to its smaller subproblems.

❌ Step 5: Exclude Item (If Too Heavy)

```
else
    dp[i][w] = dp[i - 1][w];
```

- If the item's weight exceeds the current capacity `w`,
  → cannot include it → take the result from previous item (`i-1`).

✅ Step 6: Final Answer

```
return dp[n][capacity];
```

- The last cell `dp[n][capacity]` contains the **maximum value** achievable with all `n` items and the full knapsack capacity.

🚀 Main Method

```
public static void main(String[] args) {
```

- Entry point for program execution.

```
int value[] = {60, 100, 120};
int weight[] = {10, 20, 30};
int capacity = 50;
int n = value.length;
```

- Input arrays:
    - `value[]`: values (profits) of items.
    - `weight[]`: weights of items.
    - `capacity`: total capacity of knapsack.
- `n` = number of items.

```
System.out.println("Maximum value in Knapsack = " + knapSack(capacity, weight, value, n));
```

- Calls the `knapSack()` function.

- Prints the **maximum achievable value**.

□ Dry Run Example

Input:

```
value[] = {60, 100, 120}
weight[] = {10, 20, 30}
capacity = 50
```

DP Table (Partial View):

i\w	0	10	20	30	40	50
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160	180	220

☞ Final answer = **220**

Explanation:

- Pick items with weight 20 and 30 → total weight = 50 → total value = 100 + 120 = **220**

□ Conceptual Understanding

Concept	Explanation
Problem Type	Optimization (maximize value under weight constraint)
Approach	**Dynamic Programming (Bottom-Up)**
Subproblem	dp[i][w] — max value using first i items and capacity w
Recurrence Relation	dp[i][w] = max(value[i−1] + dp[i−1][w−weight[i−1]], dp[i−1][w])
Base Case	dp[0][w] = dp[i][0] = 0
Overlapping Subproblems	Many subproblems (like smaller capacities) are reused multiple times
Optimal Substructure	Each optimal solution is built from optimal subproblems

---

⚖□ 0/1 Knapsack vs Fractional Knapsack

Feature	0/1 Knapsack	Fractional Knapsack
Approach	Dynamic Programming	Greedy Algorithm
Can take fractions?	✕ No (either take or leave)	✓ Yes
Structure Used	2D DP table	Sorting + Iteration

Feature	0/1 Knapsack	Fractional Knapsack
Time Complexity	O(n × capacity)	O(n log n)
Optimal Substructure	Yes	Yes
Overlapping Subproblems	Yes	No
Real-life Example	Packing discrete objects (laptop, books)	Filling liquid, gold dust, etc.

□ Complexity Analysis

Type	Complexity
Time	O(n × capacity) — nested loops
Space	O(n × capacity) — 2D DP table
Optimized Space	Can be reduced to O(capacity) using 1D array optimization

---

□ Connections to Subject Areas

Subject	Concept Used
Dynamic Programming	Tabulation (bottom-up approach)
Data Structures	2D arrays
Algorithm Design	Optimal Substructure, Overlapping Subproblems
Complexity Analysis	Big O Time/Space trade-offs
Mathematical Optimization	Maximization under constraint

⚡ Possible Extensions / Improvements

- □ Use **space optimization** (1D DP array) → reduces memory from O(n×W) to O(W).
- 💬 Add **user input** (like in Fractional Knapsack) for interactivity.
- 📈 Visualize DP table for better understanding (can be printed or displayed as grid).

```
public class NQueensSimple {

static final int N = 4; // You can change N to 8, etc.

// Function to print the board

static void printBoard(int board[][]) {

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

System.out.print((board[i][j] == 1 ? "Q " : ". "));

}

System.out.println();

}

System.out.println();

}

// Function to check if placing a queen is safe

static boolean isSafe(int board[][], int row, int col) {

// Check this row on the left

for (int i = 0; i < col; i++)

if (board[row][i] == 1)

return false;

// Check upper diagonal

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j] == 1)

return false;

// Check lower diagonal

for (int i = row, j = col; i < N && j >= 0; i++, j--)

if (board[i][j] == 1)

return false;

return true;

}
```

```
// Recursive function to solve N-Queens problem

static boolean solveNQ(int board[][], int col) {

if (col >= N)

return true;

for (int i = 0; i < N; i++) {

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveNQ(board, col + 1))

return true;

// Backtrack

board[i][col] = 0;

}

}

return false;

}

public static void main(String[] args) {

int board[][] = new int[N][N];

// Try placing the first queen in the first column (user can change position)

board[0][0] = 1;

// Start solving from the next column

if (solveNQ(board, 1)) {

System.out.println("Final N-Queens Board:");

printBoard(board);

} else {

System.out.println("No solution found with first Queen at (0,0).");

System.out.println("Trying different first positions...");

// Try placing the first Queen in other rows if needed

boolean found = false;
for (int i = 0; i < N; i++) {

board = new int[N][N];

board[i][0] = 1;

if (solveNQ(board, 1)) {

System.out.println("Solution found with first Queen at row " + i + ":");

printBoard(board);

found = true;

break;

}

}

if (!found)

System.out.println("Solution does not exist for any first position!");

}

}

}
```

□ Code Explanation (Line by Line)

📘 Class Declaration

```
public class NQueensSimple {
```

- Declares a public class named `NQueensSimple`.
- The problem: Place `N` queens on an `N×N` chessboard such that **no two queens attack each other**.

⚙️ Constant Declaration

```
static final int N = 4; // You can change N to 8, etc.
```

- `N` = size of the chessboard (and number of queens).
- Here, we're solving for **4-Queens**, but it can be changed to **8**, **5**, etc.
- Declared as `final` to make it constant.

🎨 Function to Print the Board

```
static void printBoard(int board[][]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            System.out.print((board[i][j] == 1 ? "Q " : ". "));
        }
        System.out.println();
    }
    System.out.println();
}
```

□ Explanation:

- The board is a 2D array (size N×N):
  - 1 → Queen is placed.
  - 0 → Empty square.
- The nested loop prints:
  - "Q " for a queen.
  - ". " for an empty cell.
- A line break (System.out.println()) after each row to make it look like a chessboard.

🎹 Output Example (for N=4):

```
. Q . .
. . . Q
Q . . .
. . Q .
```

⚔□ Check if a Position is Safe

```
static boolean isSafe(int board[][], int row, int col) {
```

- Checks if it's **safe** to place a queen at position (row, col) — i.e., no other queen can attack it.

□ 1□ Check Row on the Left

```
for (int i = 0; i < col; i++)
    if (board[row][i] == 1)
        return false;
```

- Scans the **left side** of the same row to see if any queen is already placed.
- We don't check the right side because queens are placed **column by column**, left to right.

□ 2□ Check Upper Diagonal (Top-Left Direction)

```
for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
    if (board[i][j] == 1)
        return false;
```

- Moves **up-left** diagonally from (row, col) to see if there's another queen.

□ 3□ Check Lower Diagonal (Bottom-Left Direction)

```
for (int i = row, j = col; i < N && j >= 0; i++, j--)
    if (board[i][j] == 1)
        return false;
```

- Moves **down-left** diagonally from (row, col) checking for any queen.

✅ *If All Checks Pass*

```
return true;
```

- Returns true → safe to place queen here.

□ **Concept:**
This function ensures that **no two queens share the same row, column, or diagonal** — the core rule of the N-Queens problem.

🔁 Recursive Function: Solving N-Queens

```
static boolean solveNQ(int board[][], int col) {
```

- Solves the problem **column by column** using recursion and backtracking.
- `col` → current column we're trying to place a queen in.

🏁 *Base Case*

```
if (col >= N)
    return true;
```

- If we've successfully placed queens in all columns (0 to N−1), we've found a valid configuration.
- Return `true` to indicate success.

♟□ *Try Placing a Queen in Each Row*

```
for (int i = 0; i < N; i++) {
    if (isSafe(board, i, col)) {
        board[i][col] = 1;
```

- For the current column `col`, try every row `i`.
- If it's safe to place a queen at `(i, col)`, place it (`board[i][col] = 1`).

🔁 *Recurse to Place Next Queen*

```
if (solveNQ(board, col + 1))
    return true;
```

- Recursively try to place the next queen in the next column.
- If successful, return true → stop further searching.

🔙 *Backtrack (Undo Step)*

```
board[i][col] = 0;
```

- If placing a queen at `(i, col)` didn't lead to a solution, **remove it** and try the next row.
- This is the **backtracking step**.

□ **Concept:**
Backtracking systematically explores all possibilities and **"undoes"** wrong moves when a dead end is reached.

⊘ No Solution Found in This Column

```
return false;
```

- If no position in this column works, return false → triggers backtracking to the previous column.

🚀 Main Method

```
public static void main(String[] args) {
    int board[][] = new int[N][N];
```

- Creates an empty chessboard filled with 0s (no queens yet).

🎯 Start with First Queen at (0, 0)

```
board[0][0] = 1;
```

- Initially places the first queen manually at top-left corner.

□ Start Solving from Next Column

```
if (solveNQ(board, 1)) {
    System.out.println("Final N-Queens Board:");
    printBoard(board);
}
```

- Starts the recursion from column 1 (since column 0 already has a queen).
- If a valid solution is found → print the final board.

🔁 If No Solution, Try Other Starting Rows

```
else {
    System.out.println("No solution found with first Queen at (0,0).");
    System.out.println("Trying different first positions...");
```

- If the initial position doesn't yield a solution, tries other starting positions for the first queen.

🔁 Try Each Possible Starting Row

```
boolean found = false;
for (int i = 0; i < N; i++) {
    board = new int[N][N];
    board[i][0] = 1;
    if (solveNQ(board, 1)) {
        System.out.println("Solution found with first Queen at row " + i + ":");
        printBoard(board);
        found = true;
        break;
    }
}
```

- Loops over each row in column 0:
  - Resets the board.
  - Tries placing the first queen at `(i, 0)`.

    - Runs the backtracking algorithm again.
    - Stops if a solution is found.

⊘ *If Still No Solution*

```
if (!found)
    System.out.println("Solution does not exist for any first position!");
```

- If all possible starting rows fail, print no solution.

☐ Conceptual Breakdown

Concept	Explanation
Problem Type	Constraint satisfaction problem
Approach	Backtracking (recursive trial and error)
Base Case	All queens placed (col == N)
Decision Space	Each column → try each row
Backtracking Step	Remove queen when configuration fails
Safety Check	Ensures no two queens attack each other
Search Space	N^n possible configurations
Optimization	Pruning unsafe branches early (using isSafe)

☐ Complexity Analysis

Type	Complexity
Time Complexity	O(N!) — for each column, try N possible rows.
Space Complexity	$O(N^2)$ — for the board, or O(N) if we optimize.
Auxiliary Space	O(N) — recursion stack depth.

☐ Relation to Computer Science Subjects

Subject	Concept Applied
Data Structures	2D Arrays, Recursion Stack
Algorithms	Backtracking, Constraint Checking
AI / Problem Solving	State space search (like in CSP problems)
Mathematics / Combinatorics	Permutations of queens across N columns
Complexity Theory	Non-polynomial (NP) type problem

☐ Dry Run (N = 4)

Step-by-step solution:

1. Place queen at (0, 0).
2. Next column → try safe position:
    - (1,1) ❌ (diagonal clash)
    - (2,1) ✅ → place at (2,1)
3. Next column (2):
    - (0,2) ❌
    - (1,2) ❌
    - (3,2) ✅ → place at (3,2)
4. Next column (3):
    - (0,3) ✅ → place at (0,3)
    - 🎉 All columns filled → solution found.

Output board:

```
. . Q .
Q . . .
. . . Q
. Q . .
```

⚡ Real-World Relevance

- Used in **chess AI** and **constraint-solving systems**.
- Fundamental example in **AI search algorithms**, **recursion**, and **backtracking**.
- The technique is also applied to:
    - Sudoku solvers
    - Crossword puzzle filling
    - Graph coloring
    - Scheduling problems

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

/// @title Simple bank account per-user contract (single contract, many users)
/// @notice Users can deposit ETH, withdraw up to their balance, and view balances
/// @dev Uses events, checks, and basic protections (nonReentrant not used to keep simple; for production add
ReentrancyGuard)
contract BankAccount {
    // mapping of user address => balance in wei
    mapping(address => uint256) private balances;

    // events
    event Deposit(address indexed user, uint256 amount, uint256 newBalance);
    event Withdraw(address indexed user, uint256 amount, uint256 newBalance);

    /// @notice Deposit ETH into caller's account
    /// @dev payable function, value is added to caller's balance
    function deposit() external payable {
        require(msg.value > 0, "Deposit: amount must be > 0");
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value, balances[msg.sender]);
    }

    /// @notice Withdraw `amount` wei from caller's account
    /// @param amount amount to withdraw in wei
    function withdraw(uint256 amount) external {
        require(amount > 0, "Withdraw: amount must be > 0");
        uint256 bal = balances[msg.sender];
        require(bal >= amount, "Withdraw: insufficient balance");

        // Effects
        unchecked { balances[msg.sender] = bal - amount; }

        // Interaction
        (bool sent, ) = payable(msg.sender).call{value: amount}("");
        require(sent, "Withdraw: failed to send Ether");

        emit Withdraw(msg.sender, amount, balances[msg.sender]);
    }

    /// @notice Get balance of caller (wei)
    function getMyBalance() external view returns (uint256) {
        return balances[msg.sender];
    }

    /// @notice Get balance of an arbitrary user (public read)
    function getBalanceOf(address user) external view returns (uint256) {
        return balances[user];
    }

    /// @notice Allow contract to receive plain ETH (maps to msg.sender balance)
    receive() external payable {
        // treat this like deposit
        require(msg.value > 0, "receive: no value");
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value, balances[msg.sender]);
    }

    /// @notice fallback function (called when calldata is non-empty or function doesn't exist)
    fallback() external payable {
        // fallback will also credit sender so they can send funds with data
        if (msg.value > 0) {
            balances[msg.sender] += msg.value;
            emit Deposit(msg.sender, msg.value, balances[msg.sender]);
        }
    }
}
```

Full Line-by-Line Explanation

1️⃣ License Identifier and Compiler Version

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
```

- **// SPDX-License-Identifier: MIT**
  → Required by Solidity to specify license (important for open-source verification on Etherscan).
  MIT is permissive — others can reuse your code with attribution.

- **pragma solidity ^0.8.17;**
  → Sets the **compiler version** (≥ 0.8.17, < 0.9.0).
  Version 0.8+ automatically includes **integer overflow/underflow protection**, improving safety.

2️⃣ Contract Declaration & Documentation

```
/// @title Simple bank account per-user contract (single contract, many users)
/// @notice Users can deposit ETH, withdraw up to their balance, and view balances
/// @dev Uses events, checks, and basic protections (nonReentrant not used to keep simple; for production add ReentrancyGuard)
contract BankAccount {
```

- Solidity supports **NatSpec comments (///)**, which document your contract for dev tools and users (like Doxygen for Solidity).
  - @title — contract name
  - @notice — what users should know
  - @dev — info for developers
- **contract BankAccount** — defines a new smart contract (similar to a class in OOP).

3️⃣ State Variable: Balances Mapping

```
mapping(address => uint256) private balances;
```

- mapping = a **key-value store** on blockchain (like a hash table).
  - Key → Ethereum address
  - Value → amount of Ether (in **wei**, the smallest ETH unit: 1 ETH = 10^{18} wei).
- **private** = can't be read directly by other contracts, but can be accessed via getter functions.

□ **Concept:**
In Solidity, all state variables are stored on-chain, so balances persists between transactions.

4️⃣ Events

```
event Deposit(address indexed user, uint256 amount, uint256 newBalance);
event Withdraw(address indexed user, uint256 amount, uint256 newBalance);
```

- **Events** are logs stored on the blockchain.
  - They make transactions searchable in explorers (like Etherscan).
  - indexed parameters can be used as filters.

Example:
If Alice deposits 1 ETH, an event log will record that so external applications (like a DApp UI) can track it.

5️⃣ Deposit Function

```
function deposit() external payable {
    require(msg.value > 0, "Deposit: amount must be > 0");
    balances[msg.sender] += msg.value;
    emit Deposit(msg.sender, msg.value, balances[msg.sender]);
}
```

Step-by-step:

1. **external payable**
   - external → callable from outside (not internal).
   - payable → allows the function to receive Ether.
2. **require(msg.value > 0, "...")**
   - Ensures positive deposit.
   - msg.value → amount of ETH (in wei) sent with this transaction.
3. **balances[msg.sender] += msg.value;**
   - Increases the sender’s stored balance.
   - msg.sender = address that called the function.
4. **Emit event**
   - Records the deposit operation publicly for transparency.

Concepts:

- This function modifies blockchain state (so it costs **gas**).
- Follows the **Checks-Effects-Interactions** pattern (safe order of operations).

---

6 Withdraw Function

```
function withdraw(uint256 amount) external {
    require(amount > 0, "Withdraw: amount must be > 0");
    uint256 bal = balances[msg.sender];
    require(bal >= amount, "Withdraw: insufficient balance");

    // Effects
    unchecked { balances[msg.sender] = bal - amount; }

    // Interaction
    (bool sent, ) = payable(msg.sender).call{value: amount}("");
    require(sent, "Withdraw: failed to send Ether");

    emit Withdraw(msg.sender, amount, balances[msg.sender]);
}
```

Explanation:

✅Step 1 — Validation

- Ensure user withdraws a valid amount and has enough balance.

Step 2 — Update State

```
unchecked { balances[msg.sender] = bal - amount; }
```

- **unchecked** disables Solidity’s automatic overflow check (safe because we already validated bal >= amount).
- Updates the balance **before** sending ETH → this is critical to avoid **reentrancy attacks**.

💸Step 3 — Send ETH

```
(bool sent, ) = payable(msg.sender).call{value: amount}("");
```

- Sends ETH using .call{value: amount}(""), the **recommended** way (modern replacement for .transfer or .send).
- Returns true/false to indicate success.
- If sending fails, transaction reverts.

□ *Step 4 — Log the Withdrawal*

- Emits Withdraw event for record keeping.

□ **Security Note:**
In production, you should add **nonReentrant modifier** from ReentrancyGuard (OpenZeppelin) to prevent attacks like *The DAO hack* (2016).

7□ Getter Functions

```
function getMyBalance() external view returns (uint256) {
    return balances[msg.sender];
}
```

- Returns caller's current ETH balance (from contract's mapping).
- **view** means it does not modify the blockchain → no gas used if called locally.

```
function getBalanceOf(address user) external view returns (uint256) {
    return balances[user];
}
```

- Same as above but lets anyone check any address's balance.

□ **Concept:**
All state on Ethereum is public anyway, but this provides a clean and readable API for DApps.

8□ receive() Function

```
receive() external payable {
    require(msg.value > 0, "receive: no value");
    balances[msg.sender] += msg.value;
    emit Deposit(msg.sender, msg.value, balances[msg.sender]);
}
```

- Special built-in function triggered when **plain ETH** is sent to the contract **without any calldata** (no function call).
- Treats it like a normal deposit.
- Useful for address(this).transfer() calls or direct sends from wallets.

□ **Concept:**

- Each contract can have **one receive()** function.
- If not defined, such ETH transfers will **fail** unless fallback() is payable.

9□ fallback() Function

```
fallback() external payable {
    if (msg.value > 0) {
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value, balances[msg.sender]);
    }
}
```

- Invoked when:
  - A function call **does not match any existing function** name, or
  - ETH is sent **with data**.
- This fallback also **credits ETH** sent with unexpected data.

□ **Concept:**
Together, `receive()` and `fallback()` make your contract **robust** to receive ETH under any condition.

□ Key Solidity Concepts Used

Concept	Description
State Variables	Persist between transactions, stored in blockchain state.
Mappings	Gas-efficient key-value storage (not iterable).
Payable Functions	Allow ETH transfers into the contract.
Events	Logs stored on blockchain, not accessible to smart contracts but used for UI.
Modifiers	`view`, `external`, `payable`, `private`, etc. control visibility & gas usage.
Checks-Effects-Interactions Pattern	Prevents reentrancy attacks by updating state before sending ETH.
Fallback and Receive	Handle unexpected ETH and data transfers.
msg.sender / msg.value	Global variables: transaction sender and ETH amount.

⚙□ Subject Relevance

Subject	Concept Applied
Blockchain Technology	Smart contract logic, Ether handling
Security in Distributed Systems	Reentrancy protection, input validation
Object-Oriented Programming	Encapsulation (private data + public functions)
Data Structures	Mapping for constant-time balance lookup
Software Engineering	Event logging, modular design
Cryptoeconomics	Balance tracking, incentive security

□ Complexity Analysis

Operation	Time	Gas Cost
Deposit	O(1)	Low (~21k gas)
Withdraw	O(1)	Moderate (due to ETH transfer)
Get Balance	O(1)	Free (off-chain)

☐ Security & Best Practices

Risk	Protection / Suggestion
Reentrancy	Add ReentrancyGuard (OpenZeppelin)
Unchecked Math	Safe due to require() before use
Denial of Service via Fallback	Use .call (done here ✅)
Unrestricted Access	Only affects own balance — safe
Missing onlyOwner functions	Not needed (multi-user design)

💡 Example Flow

Action	Caller	ETH Sent	Result
deposit()	Alice	1 ETH	balances[Alice] = 1
deposit()	Bob	0.5 ETH	balances[Bob] = 0.5
withdraw(0.3 ETH)	Alice	—	Sends 0.3 ETH back, balance = 0.7
getMyBalance()	Bob	—	Returns 0.5 ETH

☐ On-Chain Storage View

Address	Balance (wei)
0xAlice...	700000000000000000
0xBob...	500000000000000000

✅ Summary

This BankAccount contract demonstrates:

- **Mapping-based per-user storage**
- **Safe ETH deposit and withdrawal**
- **Event-driven design**
- **Separation of logic and safety checks**
- **Receive/fallback for full ETH compatibility**

```
}

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract StudentData {
    // Structure for Student
    struct Student {
        uint256 id;
        string name;
        uint256 age;
        string course;
    }

    // Array to store student data
    Student[] public students;

    // Mapping to check if a student exists by ID
    mapping(uint256 => bool) public studentExists;

    // Add new student
    function addStudent(uint256 _id, string memory _name, uint256 _age, string memory _course) public {
        require(!studentExists[_id], "Student with this ID already exists!");
        students.push(Student(_id, _name, _age, _course));
        studentExists[_id] = true;
    }

    // Get student details by ID
    function getStudent(uint256 _id) public view returns (string memory, uint256, string memory) {
        for (uint256 i = 0; i < students.length; i++) {
            if (students[i].id == _id) {
                return (students[i].name, students[i].age, students[i].course);
            }
        }
        revert("Student not found");
    }

    // Get total number of students
    function getTotalStudents() public view returns (uint256) {
        return students.length;
    }

    // Update student course
    function updateCourse(uint256 _id, string memory _newCourse) public {
        for (uint256 i = 0; i < students.length; i++) {
            if (students[i].id == _id) {
                students[i].course = _newCourse;
                return;
            }
        }
        revert("Student not found");
    }

    // Fallback function
    fallback() external payable {
        // Triggered when contract receives Ether or invalid function call
    }

    // Receive Ether function
    receive() external payable {}
}
```

Subject Context

This contract is a **Solidity-based decentralized application (DApp) backend** — it defines data (students) and logic (functions) to **store, update, and retrieve** student records on a **blockchain**.

Main topics involved:

- **Solidity language basics** (types, structs, arrays, mappings, modifiers)
- **State variables & storage**
- **Functions & visibility**
- **Smart contract deployment/execution**
- **Fallback and receive functions**
- **Gas optimization and security**

💻 Code Explanation (Line-by-Line)

```
// SPDX-License-Identifier: MIT
```

✅ **SPDX License Identifier**
Specifies that this contract uses the MIT license — required for open-source clarity.

```
pragma solidity ^0.8.0;
```

✅ **Compiler Directive**
Ensures this code compiles with **Solidity version 0.8.0 or higher**, which includes built-in **overflow protection**.

```
contract StudentData {
```

✅ **Contract Declaration**
Defines a **smart contract** named `StudentData`.
Think of it as a **class in OOP**, deployed on the blockchain.

```
    struct Student {
        uint256 id;
        string name;
        uint256 age;
        string course;
    }
```

✅ **Struct Definition**
A `struct` groups multiple related data fields — similar to a **C struct** or a **Java class without methods**.
Each `Student` has:

- `id` → unique identifier
- `name` → student's name
- `age` → student’s age
- `course` → name of enrolled course

📃 **Concept:**
Used for **custom data types** and represents how we model real-world entities (students).

```
    Student[] public students;
```

✅ **Dynamic Array**
Stores multiple `Student` structs.
`public` creates an automatic **getter function** so anyone can read data.

```
    mapping(uint256 => bool) public studentExists;
```

✅ **Mapping** (Key-Value Store)

- Key: `student ID`
- Value: Boolean indicating if student exists
  This helps in **constant-time lookups** for existence checks.

📃 **Concept:**
mapping is like a **hash map** — ideal for quick checks without looping.

□ Function 1: addStudent

```
function addStudent(uint256 _id, string memory _name, uint256 _age, string memory _course) public {
    require(!studentExists[_id], "Student with this ID already exists!");
    students.push(Student(_id, _name, _age, _course));
    studentExists[_id] = true;
}
```

✅ **Purpose:** Adds a new student to the blockchain record.

Line-by-line:

1. **Function parameters:** _id, _name, _age, _course
2. require(!studentExists[_id], ...) → prevents duplicate IDs.
3. students.push(...) → adds the student to the array.
4. studentExists[_id] = true; → marks this ID as taken.

📃 **Concepts:**

- require() → validation check; reverts if false.
- memory keyword → temporary data stored only during execution.

□ Function 2: getStudent

```
function getStudent(uint256 _id) public view returns (string memory, uint256, string memory) {
    for (uint256 i = 0; i < students.length; i++) {
        if (students[i].id == _id) {
            return (students[i].name, students[i].age, students[i].course);
        }
    }
    revert("Student not found");
}
```

✅ **Purpose:** Retrieve student details by their ID.

Explanation:

- Iterates through all students.
- If match found, returns (name, age, course).
- If not found, revert stops execution and refunds unused gas.

📃 **Concepts:**

- view → indicates this function doesn't modify the blockchain (no gas cost when called externally).
- revert() → throws an error with a message.

□ Function 3: getTotalStudents

```
function getTotalStudents() public view returns (uint256) {
    return students.length;
}
```

✅ **Purpose:** Returns total number of student entries.
Simple read-only function.

📘 **Concept:**
Helps estimate **storage usage** or for listing purposes.

□ Function 4: `updateCourse`

```
function updateCourse(uint256 _id, string memory _newCourse) public {
    for (uint256 i = 0; i < students.length; i++) {
        if (students[i].id == _id) {
            students[i].course = _newCourse;
            return;
        }
    }
    revert("Student not found");
}
```

✅ **Purpose:** Updates a student's course by their ID.

📘 **Concepts:**

- `memory` used for `_newCourse` means it's not stored permanently unless assigned.
- Uses **loop + conditional** logic to locate and modify data.

⚠□ **Gas Note:**
Linear search (`O(n)`) can be costly on large datasets.
You can optimize by storing `mapping(uint => Student)` instead of an array.

□ Fallback & Receive Functions

```
fallback() external payable {
    // Triggered when contract receives Ether or invalid function call
}

receive() external payable {}
```

✅ **Purpose:** Handle ETH transfers and unexpected calls.

Difference:

- `receive()` → called when plain ETH sent with empty data.
- `fallback()` → called when invalid function or non-empty data sent.

📘 **Concepts:**

- Both are `payable`, so they can **receive Ether**.
- Useful for **donations**, **custom payments**, or **defensive programming**.

⚙ Subject Connections

Concept	Subject	Explanation
struct, array, mapping	**Data Structures**	Used for efficient data organization and lookup.
require, revert	**Error Handling**	Ensures validity and prevents state corruption.
public, view, memory	**Solidity Keywords**	Control visibility, state changes, and memory usage.
fallback, receive	**Blockchain Interaction**	Handle ETH and unknown function calls.
gas, state, transaction	**Ethereum Blockchain**	Execution costs and immutable data storage.

Summary

Function	Purpose	Key Concept
addStudent	Adds new student	Structs, mapping, validation
getStudent	Retrieves details	Loops, return tuples
getTotalStudents	Counts entries	Array length
updateCourse	Edits data	Loop search & update
fallback / receive	Handle ETH	Blockchain interaction

```
# BANK CUSTOMER CHURN PREDICTION USING NEURAL NETWORK
# Dataset: https://www.kaggle.com/barelydedicated/bank-customer-churn-modeling

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

# 1. Read the dataset
df = pd.read_csv("Churn_Modelling.csv")  # Change path if needed
print("Dataset preview:")
print(df.head())

# Drop irrelevant columns
df = df.drop(['RowNumber', 'CustomerId', 'Surname'], axis=1)

# 2. Distinguish features and target
X = df.drop('Exited', axis=1)   # Features
y = df['Exited']                # Target

# Encode categorical variables (Geography, Gender)
label_encoder = LabelEncoder()
X['Gender'] = label_encoder.fit_transform(X['Gender'])  # Male=1, Female=0
X = pd.get_dummies(X, columns=['Geography'], drop_first=True)  # One-hot encode

# 3. Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Normalize data
```

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# 4. Initialize and build the model
model = Sequential()
model.add(Dense(16, input_dim=X_train.shape[1], activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(8, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2, verbose=0)

# Plot accuracy
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.title('Model Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.show()

# 5. Evaluate Model
y_pred = (model.predict(X_test) > 0.5).astype("int32")

# Print accuracy and confusion matrix
acc = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

print("\n✅ Model Evaluation Results:")
print(f"Accuracy: {acc:.4f}")
print("\nConfusion Matrix:")
print(cm)
print("\nClassification Report:")

print(classification_report(y_test, y_pred))


# Visualize confusion matrix

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.show()
```

□ Project Overview

Goal:
Predict whether a customer will **leave the bank (churn)** or **stay**, based on features like credit score, age, balance, etc.

Dataset:
Bank Customer Churn Modeling

Approach:
Use a **Neural Network (ANN)** to perform **binary classification** (`Exited` = 1 means churned, 0 means stayed).

💻 Code Breakdown (Line by Line)

1□ Importing Libraries

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
```

✓ Concepts:

- **pandas, numpy** → for data handling and numerical computation.
- **seaborn, matplotlib** → for visualization.
- **scikit-learn** → for preprocessing, splitting data, and evaluation.
- **tensorflow.keras** → for building and training the Artificial Neural Network.

□ Related Subjects:

- *Machine Learning Foundations*

- *Deep Learning*
- *Data Preprocessing*

2️⃣ Read and Inspect Dataset

```
df = pd.read_csv("Churn_Modelling.csv")
print("Dataset preview:")
print(df.head())
```

✅ Loads the dataset from a .csv file into a pandas DataFrame and shows the first 5 rows.

3️⃣ Drop Irrelevant Columns

```
df = df.drop(['RowNumber', 'CustomerId', 'Surname'], axis=1)
```

✅ Removes identifiers that don't influence churn:

- These are **non-predictive** columns.
- Dropping them avoids noise in training.

📘 **Concept:**
Feature selection — removing unnecessary attributes improves generalization.

4️⃣ Separate Features and Target

```
X = df.drop('Exited', axis=1)
y = df['Exited']
```

✅

- X: All **independent features** (inputs).
- y: **Target label** (Exited).

📘 **Concept:**
This is a **supervised learning** setup — the model learns to map inputs → output label.

5️⃣ Encode Categorical Variables

```
label_encoder = LabelEncoder()
X['Gender'] = label_encoder.fit_transform(X['Gender'])  # Male=1, Female=0
X = pd.get_dummies(X, columns=['Geography'], drop_first=True)
```

✅ Steps:

- LabelEncoder converts text labels into numbers (for gender).
- get_dummies() performs **one-hot encoding** for "Geography" (France, Germany, Spain → binary columns).

📘 **Concept:**
Neural networks can only process numerical data, not text — so encoding converts categories to numeric format.

6️⃣ Train-Test Split

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

✅ Splits data into:

- 80% for training the model
- 20% for testing it later

📘 **Concept:**
Used to **evaluate model generalization** on unseen data.

7️⃣ Feature Scaling

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

✅ Normalizes all feature values (mean = 0, std = 1).

📘 **Concept:**
Neural networks perform better when all input features are on a similar scale — avoids bias toward larger numeric ranges.

8️⃣ Build Neural Network Model

```
model = Sequential()
model.add(Dense(16, input_dim=X_train.shape[1], activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(8, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))
```

✅ Defines a **feedforward neural network** (Multi-Layer Perceptron).

Layer-by-layer explanation:

- `Dense(16, activation='relu')`:
  Hidden layer with 16 neurons, ReLU activation (rectified linear unit).
- `Dropout(0.3)`:
  Randomly drops 30% of neurons to prevent **overfitting**.
- Another hidden layer (8 neurons).
- Output layer:
  `Dense(1, activation='sigmoid')` → outputs value between 0 and 1 (probability of churn).

🧾 **Concepts:**

- **ReLU** → avoids vanishing gradient; efficient for hidden layers.
- **Sigmoid** → ideal for binary classification.
- **Dropout** → regularization technique.
- **Sequential Model** → layer-by-layer stack.

9️⃣ Compile the Model

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

✅ Defines:

- **Optimizer:** `adam` (adaptive learning rate optimizer)
- **Loss function:** `binary_crossentropy` for binary outcomes
- **Metric:** Accuracy

🧾 **Concept:**
Compilation defines **how the model learns** — optimizer updates weights, loss quantifies error.

🔟 Train the Model

```
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2, verbose=0)
```

✅ Trains for 50 iterations (epochs), using:

- 80% training data, 20% for validation
- Batch size = 32 (processes 32 samples at a time)

🧾 **Concepts:**

- **Epoch:** One full pass over the training data.
- **Batch:** Subset processed before weight update.
- **Validation Split:** Helps monitor overfitting during training.

📊 Plot Model Accuracy

```
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.title('Model Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.show()
```

✅ Visualizes how accuracy changes over time (to detect overfitting or underfitting).

🔍 Evaluate Model

```
y_pred = (model.predict(X_test) > 0.5).astype("int32")
```

✅ Predicts probabilities and converts to binary (1 = churn, 0 = stay).

📈 Accuracy and Confusion Matrix

```
acc = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

print("\n✅ Model Evaluation Results:")
print(f"Accuracy: {acc:.4f}")
print("\nConfusion Matrix:")
print(cm)
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

✅ Evaluation metrics:

- **Accuracy:** % of correct predictions
- **Confusion Matrix:** TP, FP, TN, FN counts
- **Classification Report:** Precision, Recall, F1-score

📒 **Concepts:**

- **Precision:** How many predicted churns are correct.
- **Recall:** How many actual churns are caught.
- **F1 Score:** Balance between precision and recall.

📉 Visualize Confusion Matrix

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

✅ Creates a heatmap for intuitive understanding of model performance.

Summary Table

Step	Concept	Purpose
Data Loading	Pandas	Import dataset
Cleaning	Feature Selection	Drop irrelevant columns
Encoding	Label Encoding, One-hot	Convert categorical → numeric
Scaling	Standardization	Normalize data

Step	Concept	Purpose
Model	Neural Network (ANN)	Predict churn
Training	Backpropagation	Adjust weights
Evaluation	Accuracy, Confusion Matrix	Measure performance

⚙️ Key Deep Learning Concepts Used

- **Feedforward Neural Network**
- **ReLU & Sigmoid activations**
- **Dropout Regularization**
- **Binary Crossentropy Loss**
- **Adam Optimizer**
- **Batch Training**
- **Validation Curve Analysis**

📚 Subject Linkage

Subject	Related Concept
Artificial Intelligence	Predictive modeling
Machine Learning	Classification algorithms
Deep Learning	Neural Networks (ANN)
Data Science	Data preprocessing, feature engineering
Statistics	Evaluation metrics (Precision, Recall)

```
# Diabetes Prediction using K-Nearest Neighbors (KNN)
# Dataset: https://www.kaggle.com/datasets/abdallamahgoub/diabetes

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (
    confusion_matrix,
    accuracy_score,
    precision_score,
    recall_score
)

# 1. Load dataset
df = pd.read_csv("diabetes.csv")  # Ensure the CSV is in the same folder
print("Dataset preview:")
print(df.head())

# 2. Check basic info
print("\nDataset info:")
print(df.info())

# 3. Separate features and target
X = df.drop('Outcome', axis=1)  # Features
y = df['Outcome']               # Target (1 = Diabetic, 0 = Non-Diabetic)

# 4. Split into training and test data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

```
# 5. Normalize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# 6. Initialize and train KNN model
k = 7  # you can tune this value
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)

# 7. Make predictions
y_pred = knn.predict(X_test)

# 8. Evaluate model
cm = confusion_matrix(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
error_rate = 1 - accuracy

# 9. Print results
print("\n✅ Model Evaluation Results:")
print(f"Confusion Matrix:\n{cm}")
print(f"Accuracy: {accuracy:.4f}")
print(f"Error Rate: {error_rate:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")

# 10. Visualize Confusion Matrix
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title(f'KNN (k={k}) - Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

# 11. (Optional) Tune k value for best accuracy
error_rates = []
```

```
for i in range(1, 21):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train, y_train)
    y_pred_i = knn.predict(X_test)
    error_rates.append(1 - accuracy_score(y_test, y_pred_i))

plt.plot(range(1, 21), error_rates, marker='o', linestyle='dashed')
plt.title('Error Rate vs K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
plt.show()
```

□ Project: Diabetes Prediction using KNN

Goal:
Predict whether a person is diabetic (Outcome = 1) or not (Outcome = 0) using medical measurements such as glucose level, BMI, insulin, etc.

Dataset:
Kaggle - Diabetes Dataset

Algorithm Used:
K-Nearest Neighbors (KNN) — a **non-parametric**, **instance-based** learning algorithm.

💻 Line-by-Line Explanation

1□ Importing Libraries

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (
    confusion_matrix,
    accuracy_score,
    precision_score,
    recall_score
)
```

✅ What's happening:

- pandas, numpy → Data manipulation and numerical computing
- seaborn, matplotlib → Visualization
- train_test_split → Splits dataset into training and testing subsets
- StandardScaler → Normalizes feature values

- KNeighborsClassifier → Machine learning model for classification
- confusion_matrix, accuracy_score, precision_score, recall_score → Evaluate performance

📘 **Concepts:**

- **Supervised learning:** Using labeled data (Outcome) to train model.
- **KNN algorithm:** Classifies a data point based on the *majority vote* of its k nearest neighbors.

2️⃣ Load Dataset

```
df = pd.read_csv("diabetes.csv")
print("Dataset preview:")
print(df.head())
```

✅ Reads the dataset into a DataFrame and displays first few rows.
This step confirms that the file is loaded correctly and columns are as expected.

📘 **Concept:**
Exploratory Data Analysis (EDA) always begins by inspecting the data.

3️⃣ Dataset Info

```
print("\nDataset info:")
print(df.info())
```

✅ Displays data types and non-null counts — helps identify:

- Missing values
- Numerical vs categorical columns

4️⃣ Separate Features and Target

```
X = df.drop('Outcome', axis=1)  # Features
y = df['Outcome']               # Target (1 = Diabetic, 0 = Non-Diabetic)
```

✅

- X → All input features (like glucose, insulin, BMI, etc.)
- y → Output label (whether diabetic or not)

📘 **Concept:**
In **supervised classification**, we separate the **independent variables** (features) and **dependent variable** (target label).

5️⃣ Split into Train and Test Sets

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

✅ Splits data:

- 80% → training the model
- 20% → testing the model

`random_state=42` ensures reproducibility.

📘 **Concept:**
Used to evaluate model’s performance on **unseen data**.

6️⃣ Normalize Features

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

✅ Standardizes all feature columns (mean = 0, standard deviation = 1).

📘 **Concept:**

- KNN is a **distance-based** algorithm (uses Euclidean distance).
- Features with large ranges (e.g., “Glucose” vs “Pregnancies”) can dominate distance calculations.
- Normalization ensures **fair contribution** from all features.

7️⃣ Initialize and Train the KNN Model

```
k = 7  # you can tune this value
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)
```

✅ Creates and trains a KNN model with `k=7` neighbors.

📘 **Concepts:**

- **KNN (K-Nearest Neighbors)** works by:
  1. Measuring distance (usually Euclidean) from a query point to all training points.
  2. Picking the `k` nearest samples.
  3. Predicting the class that is most common among those neighbors.
- **Hyperparameter (k):** Controls bias-variance tradeoff:
  - Small `k` → more variance, can overfit.
  - Large `k` → more bias, may underfit.

8️⃣ Make Predictions

```
y_pred = knn.predict(X_test)
```

✅ Predicts whether each test instance is diabetic (1) or not (0).

📘 **Concept:**
The model uses learned patterns (distances from training data) to assign labels to new samples.

9️⃣ Evaluate the Model

```
cm = confusion_matrix(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
error_rate = 1 - accuracy
```

✅ Calculates key metrics.

📊 **Confusion Matrix Breakdown:**

Metric	Meaning
TP	True Positive (Predicted Diabetic = Actual Diabetic)
TN	True Negative (Predicted Non-Diabetic = Actual Non-Diabetic)
FP	False Positive (Predicted Diabetic = Actually Non-Diabetic)
FN	False Negative (Predicted Non-Diabetic = Actually Diabetic)

📘 **Concepts:**

- **Accuracy:** Overall correctness

  Accuracy=TP+TNTP+TN+FP+FN\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}Accuracy=TP+TN+FP+FNTP+TN

- **Precision:** How many predicted diabetics are actually diabetic

  Precision=TPTP+FP\text{Precision} = \frac{TP}{TP + FP}Precision=TP+FPTP

- **Recall (Sensitivity):** How many actual diabetics were correctly predicted

  Recall=TPTP+FN\text{Recall} = \frac{TP}{TP + FN}Recall=TP+FNTP

- **Error Rate:** 1−Accuracy1 - \text{Accuracy}1−Accuracy

🔟 Print Results

```
print("\n✅ Model Evaluation Results:")
print(f"Confusion Matrix:\n{cm}")
print(f"Accuracy: {accuracy:.4f}")
print(f"Error Rate: {error_rate:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
```

✅ Displays evaluation results in a readable format.

1️⃣1️⃣ Visualize Confusion Matrix

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title(f'KNN (k={k}) - Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

✅ Draws a visual heatmap — makes it easier to see correct vs incorrect predictions.

📘 **Concept:**
Visualization helps identify if the model is biased toward one class.

1️⃣2️⃣ Tune K Value

```
error_rates = []
for i in range(1, 21):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train, y_train)
    y_pred_i = knn.predict(X_test)
    error_rates.append(1 - accuracy_score(y_test, y_pred_i))

plt.plot(range(1, 21), error_rates, marker='o', linestyle='dashed')
plt.title('Error Rate vs K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
plt.show()
```

✅ Tests values of `k` from 1 to 20, calculates the error rate for each, and plots it.

📘 **Concept:**

- This is **hyperparameter tuning** (manually).
- The best `k` is usually where **error rate is minimum** and **accuracy stabilizes**.

🧠 Concept Summary

Concept	Description
KNN Algorithm	Classifies based on nearest neighbors' majority label
Distance Metric	Typically Euclidean distance
Feature Scaling	Essential since KNN depends on distances
Train-Test Split	Prevents overfitting; evaluates generalization
Confusion Matrix	Shows model prediction breakdown
Error Rate vs K Plot	Used to find optimal `k` value

📚 Subject Linkages

Subject	Concept
Machine Learning	K-Nearest Neighbors algorithm
Data Mining	Classification, distance-based learning
Statistics	Confusion matrix, precision, recall
Data Science	Preprocessing, scaling, model evaluation

☐ Key Takeaways

- KNN is **simple but powerful** for small- to medium-sized datasets.
- Always **scale features** before using KNN.
- Use **cross-validation or tuning** to select the best `k`.
- Evaluate using **recall** if missing diabetics is critical (medical use case).

```
# Email Spam Classification using KNN and SVM
# Dataset: https://www.kaggle.com/datasets/balaka18/email-spam-classification-dataset-csv

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import LabelEncoder
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# 1. Load dataset
df = pd.read_csv("emails.csv")  # Change path if needed
print("Data Preview:")
print(df.head())

# 2. Check for missing values
print("\nMissing values per column:")
print(df.isnull().sum())

# Drop any nulls
df = df.dropna()

# 3. Encode labels (spam = 1, ham = 0)
label_encoder = LabelEncoder()
df['label'] = label_encoder.fit_transform(df['spam'])  # or 'Category' column depending on dataset
df = df.rename(columns={'text': 'email_text'})

# 4. Text preprocessing – Convert text to TF-IDF vectors
tfidf = TfidfVectorizer(stop_words='english', max_features=3000)
X = tfidf.fit_transform(df['email_text']).toarray()
y = df['label']

# 5. Train-Test Split
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 6. Models
knn = KNeighborsClassifier(n_neighbors=5)
svm = SVC(kernel='linear', random_state=42)

# 7. Train Models
knn.fit(X_train, y_train)
svm.fit(X_train, y_train)

# 8. Predictions
y_pred_knn = knn.predict(X_test)
y_pred_svm = svm.predict(X_test)

# 9. Evaluation
def evaluate_model(y_true, y_pred, model_name):
    print(f"\n📊 {model_name} Results:")
    print("Accuracy:", accuracy_score(y_true, y_pred))
    print("Classification Report:\n", classification_report(y_true, y_pred))
    cm = confusion_matrix(y_true, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(f"{model_name} - Confusion Matrix")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()

evaluate_model(y_test, y_pred_knn, "K-Nearest Neighbors")
evaluate_model(y_test, y_pred_svm, "Support Vector Machine")

# 10. Comparison
acc_knn = accuracy_score(y_test, y_pred_knn)
acc_svm = accuracy_score(y_test, y_pred_svm)

print("\n✅ Performance Comparison:")
print(f"KNN Accuracy: {acc_knn:.4f}")
print(f"SVM Accuracy: {acc_svm:.4f}")
if acc_svm > acc_knn:
    print("SVM performed better overall.")
else:
    print("KNN performed better overall.")
```

📨 Project: Email Spam Classification using KNN and SVM

🎯 Objective

To classify emails as **spam (1)** or **ham (0)** using:

- **K-Nearest Neighbors (KNN)** — a distance-based algorithm
- **Support Vector Machine (SVM)** — a margin-based algorithm

📊 Dataset

Kaggle - Email Spam Classification Dataset
Contains two columns:

- text — the email content
- spam — label ("spam" or "ham")

💻 Step-by-Step Explanation with Concepts

1️⃣ Import Libraries

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import LabelEncoder
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

✅ What it does:

- pandas, numpy: handle and process the dataset
- seaborn, matplotlib: visualization
- train_test_split: splits the dataset into train/test sets
- TfidfVectorizer: converts text into numerical features (used in NLP)
- LabelEncoder: converts text labels ("spam", "ham") into numeric values
- KNeighborsClassifier & SVC: models (KNN & SVM)
- metrics: evaluate model performance

📘 Concepts:

- **Text data → numerical form** using *vectorization* (TF-IDF)
- **Supervised learning** (spam detection is a binary classification problem)
- **Model evaluation** → confusion matrix, accuracy, precision, recall, F1-score

---

2️⃣ Load Dataset

```
df = pd.read_csv("emails.csv")
print("Data Preview:")
print(df.head())
```

✅ Loads the dataset and displays the first few rows for inspection.

📘 **Concept:**
Always inspect your data first to understand structure and identify columns.

3️⃣ Check for Missing Values

```
print("\nMissing values per column:")
print(df.isnull().sum())
df = df.dropna()
```

✅ Checks for and removes missing values to prevent training errors.

📘 **Concept:**

- Missing text can lead to incomplete feature vectors.
- Dropping or imputing them ensures model integrity.

4️⃣ Encode Labels

```
label_encoder = LabelEncoder()
df['label'] = label_encoder.fit_transform(df['spam'])
df = df.rename(columns={'text': 'email_text'})
```

✅ Converts `spam` and `ham` into numerical labels:

- **spam → 1**
- **ham → 0**

📘 **Concept:**
Machine learning models can only work with **numeric** data, not text labels.

5️⃣ Text Vectorization using TF-IDF

```
tfidf = TfidfVectorizer(stop_words='english', max_features=3000)
X = tfidf.fit_transform(df['email_text']).toarray()
y = df['label']
```

✅ **TF-IDF (Term Frequency–Inverse Document Frequency)** transforms text into numerical vectors.

📘 **Concepts:**

- **TF (Term Frequency):** how often a word appears in a document
- **IDF (Inverse Document Frequency):** how unique that word is across all documents
- The TF-IDF score = TF × IDF

💡 **Why TF-IDF?**

- Common words like “the”, “is”, “and” have little value.
- TF-IDF downweights such frequent words and highlights important ones (e.g., “free”, “offer”, “win” in spam emails).

📊 **max_features=3000** limits vocabulary to the top 3,000 most informative words.

6️⃣ Split the Dataset

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

✅ 80% training data, 20% testing data
random_state=42 ensures reproducible results.

📘 **Concept:**
This prevents *data leakage* — ensures fair evaluation on unseen data.

7️⃣ Define Models

```
knn = KNeighborsClassifier(n_neighbors=5)
svm = SVC(kernel='linear', random_state=42)
```

✅ Initializes both models.

📘 **Concepts:**

Model	Concept	Key Idea
KNN	Instance-based	Classifies based on nearest neighbors
SVM	Margin-based	Finds a hyperplane that best separates classes

🔍 KNN Recap

- Computes distance (e.g., Euclidean or cosine similarity)
- Chooses the **majority label** among the nearest k samples

🔍 SVM Recap

- Tries to find the **best decision boundary (hyperplane)**
- Maximizes the **margin** (distance between the hyperplane and nearest data points — support vectors)
- kernel='linear' → good for linearly separable text data

8️⃣ Train Models

```
knn.fit(X_train, y_train)
svm.fit(X_train, y_train)
```

✅ Both models learn from the training data.

📘 **Concept:**

- **KNN** stores all training samples and classifies new data at prediction time (lazy learning).
- **SVM** calculates the optimal hyperplane during training (eager learning).

9️⃣ Make Predictions

```
y_pred_knn = knn.predict(X_test)
y_pred_svm = svm.predict(X_test)
```

✅ Models predict spam (1) or ham (0) for unseen emails.

📘 **Concept:**
Predictions are based on the patterns learned during training.

🔟 Define Evaluation Function

```
def evaluate_model(y_true, y_pred, model_name):
    print(f"\n📊 {model_name} Results:")
    print("Accuracy:", accuracy_score(y_true, y_pred))
    print("Classification Report:\n", classification_report(y_true, y_pred))
    cm = confusion_matrix(y_true, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(f"{model_name} - Confusion Matrix")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()
```

✅ Generic function that prints evaluation metrics and confusion matrix.

📘 **Concepts:**

Metric	Formula	Meaning
Accuracy	(TP+TN)/(TP+FP+FN+TN)	Overall correctness
Precision	TP/(TP+FP)	How many predicted spams are actual spams
Recall	TP/(TP+FN)	How many actual spams are detected

Metric	Formula	Meaning
F1-Score	2×(Precision×Recall)/(Precision+Recall)	Balance between Precision and Recall

---

1️⃣1️⃣ Evaluate Models

```
evaluate_model(y_test, y_pred_knn, "K-Nearest Neighbors")
evaluate_model(y_test, y_pred_svm, "Support Vector Machine")
```

✅ Displays performance metrics and heatmaps for both models.

📊 Heatmap:

- Blue squares → True predictions (Diagonal)
- Off-diagonal → Misclassifications

1️⃣2️⃣ Compare Model Performance

```
acc_knn = accuracy_score(y_test, y_pred_knn)
acc_svm = accuracy_score(y_test, y_pred_svm)

print("\n✅ Performance Comparison:")
print(f"KNN Accuracy: {acc_knn:.4f}")
print(f"SVM Accuracy: {acc_svm:.4f}")
if acc_svm > acc_knn:
    print("SVM performed better overall.")
else:
    print("KNN performed better overall.")
```

✅ Compares model accuracies to decide which one is better.

🧾 Expected Result:
Typically, **SVM** performs better in text data because:

- It handles **high-dimensional sparse data** effectively (TF-IDF has thousands of features)
- It finds a **robust linear boundary** even when data points are noisy

KNN, on the other hand, can be slower and less accurate for text data since distance metrics can struggle with sparse vectors.

📘 Subject-Wise Concept Mapping

Subject	Concept Used
Machine Learning	KNN, SVM, model evaluation
Natural Language Processing (NLP)	Text vectorization (TF-IDF), stopword removal
Data Mining	Spam detection, pattern recognition
Statistics	Precision, Recall, Confusion Matrix

Summary of Learning

Step	Concept	Importance
Preprocessing	Label encoding, TF-IDF	Converts text into numeric data
Model 1	KNN	Simple distance-based classifier
Model 2	SVM	Robust margin-based classifier
Evaluation	Accuracy, Precision, Recall	Measures model quality
Comparison	Accuracy scores	Identifies the best model

Key Takeaways

- **TF-IDF** converts emails into meaningful numerical features.
- **KNN** is intuitive but less effective for high-dimensional text.
- **SVM** is preferred for **text classification**, offering better generalization.
- Evaluation using **Precision and Recall** is vital because **false positives (ham predicted as spam)** can cause real-world issues.

```
# SALES DATA CLUSTERING USING K-MEANS AND HIERARCHICAL CLUSTERING
# Dataset: https://www.kaggle.com/datasets/kyanyoga/sample-sales-data

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster

# 1. Load dataset
df = pd.read_csv("sales_data_sample.csv", encoding='latin1')
print("Dataset preview:")
print(df.head())

# 2. Preprocessing
# Select numeric columns only for clustering
numeric_df = df.select_dtypes(include=[np.number]).dropna(axis=1, how='all')

print("\nNumeric Columns Used:")
print(numeric_df.columns.tolist())

# Scale numeric data for better clustering
scaler = StandardScaler()
scaled_data = scaler.fit_transform(numeric_df)

# 3. Determine optimal number of clusters using Elbow Method
inertia = []
K = range(1, 11)

for k in K:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(scaled_data)
    inertia.append(kmeans.inertia_)

# Plot elbow curve
```

```
plt.figure(figsize=(8, 5))
plt.plot(K, inertia, 'bo--')
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Inertia (Within-Cluster Sum of Squares)')
plt.grid(True)
plt.show()

# Based on the elbow plot, choose k (for example 3 or 4)
optimal_k = 4  # Change based on elbow visualization

# 4. Apply K-Means with chosen number of clusters
kmeans = KMeans(n_clusters=optimal_k, random_state=42, n_init=10)
df['Cluster'] = kmeans.fit_predict(scaled_data)

# 5. Visualize cluster distribution
plt.figure(figsize=(8, 5))
sns.countplot(x='Cluster', data=df, palette='viridis')
plt.title('Cluster Distribution')
plt.show()

# 6. Optional: visualize relationship between two features
if 'SALES' in numeric_df.columns and 'QUANTITYORDERED' in numeric_df.columns:
    plt.figure(figsize=(8, 6))
    sns.scatterplot(
        x=df['SALES'], y=df['QUANTITYORDERED'], hue=df['Cluster'], palette='Set2'
    )
    plt.title('Sales vs Quantity by Cluster')
    plt.show()

# 7. Hierarchical Clustering (optional visualization)
linked = linkage(scaled_data, method='ward')
plt.figure(figsize=(10, 6))
dendrogram(linked, truncate_mode='lastp', p=10, show_leaf_counts=True)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Cluster Size')
plt.ylabel('Distance')
plt.show()


print("\n✅ Clustering complete! Cluster labels added as 'Cluster' column.")

print(df[['Cluster']].head())
```

PROJECT OVERVIEW

🎯 Objective

To group similar sales data records into distinct **clusters** using:

1. **K-Means Clustering**
2. **Hierarchical Clustering**

This helps a company identify **sales patterns**, such as:

- High-value customers
- Bulk buyers
- Seasonal order groups

💻 Step-by-Step Code Explanation

---

1 Import Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
```

✅ Purpose:

- pandas, numpy: data manipulation and numerical operations
- matplotlib, seaborn: visualization tools
- StandardScaler: normalization for clustering
- KMeans: machine learning algorithm for unsupervised clustering
- dendrogram, linkage: used for **hierarchical clustering visualization**

📘 Concepts:

- **Clustering** = unsupervised learning technique to group similar data points.
- **K-Means** = partitions data into *k clusters* by minimizing intra-cluster distance.
- **Hierarchical Clustering** = builds a tree-like structure (dendrogram) showing data hierarchy.

2️⃣ Load Dataset

```
df = pd.read_csv("sales_data_sample.csv", encoding='latin1')
print("Dataset preview:")
print(df.head())
```

✅ Loads the **Sales dataset** from Kaggle.
`encoding='latin1'` handles special characters in the dataset (e.g., “São Paulo”).

📘 **Concept:**
Always inspect the dataset to check for data types, missing values, and numeric vs categorical columns.

3️⃣ Preprocessing - Select Numeric Columns

```
numeric_df = df.select_dtypes(include=[np.number]).dropna(axis=1, how='all')

print("\nNumeric Columns Used:")
print(numeric_df.columns.tolist())
```

✅ Extracts **only numeric features** because clustering algorithms rely on **distance computations**, which require numerical values.

📘 **Concept:**

- Text features (e.g., product names, countries) can’t be directly clustered without encoding.
- Non-numeric columns can later be used for **cluster interpretation**.

4️⃣ Scale Numeric Data

```
scaler = StandardScaler()
scaled_data = scaler.fit_transform(numeric_df)
```

✅ Standardizes all numeric features so that each has:

- Mean = 0
- Standard deviation = 1

📘 **Why scaling is important:**

- Without scaling, features with large magnitudes (like “SALES”) dominate the clustering.
- Distance-based algorithms like **K-Means** and **Hierarchical Clustering** are **scale-sensitive**.

📗 **Concept:**

z=x−μσz = \frac{x - \mu}{\sigma}z=σx−μ

(Standardization formula)

5️⃣ Find Optimal Number of Clusters (Elbow Method)

```
inertia = []
K = range(1, 11)

for k in K:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(scaled_data)
    inertia.append(kmeans.inertia_)
```

✅ Runs K-Means for *k = 1 to 10* and stores the **inertia** (total within-cluster sum of squares).

📘 **Concept:**

- **Inertia** = how tightly the data points are grouped within each cluster.
- Lower inertia → tighter clusters.

📉 **Elbow Method:**
Plot k vs inertia and look for the "elbow point" where inertia drops sharply — this indicates a good k.

6️⃣ Visualize Elbow Curve

```
plt.figure(figsize=(8, 5))
plt.plot(K, inertia, 'bo--')
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Inertia (Within-Cluster Sum of Squares)')
plt.grid(True)
plt.show()
```

✅ The "elbow point" visually helps decide how many clusters make sense.

📘 **Interpretation:**

- If the elbow is at **k = 3**, that's likely the best number of clusters.
- Beyond that, inertia decreases very slowly (diminishing returns).

7️⃣ Choose Optimal k and Apply K-Means

```
optimal_k = 4  # Change based on elbow visualization

kmeans = KMeans(n_clusters=optimal_k, random_state=42, n_init=10)
df['Cluster'] = kmeans.fit_predict(scaled_data)
```

✅ Applies K-Means with the chosen k (here 4) and assigns a cluster label to each record.

📘 **Concept:**

- **fit_predict()** → runs K-Means and assigns each record to the nearest cluster center.
- K-Means minimizes:

  J=∑i=1k∑x∈Ci||x−μi||2J = \sum_{i=1}^{k}\sum_{x \in C_i} ||x - \mu_i||^2J=i=1∑kx∈Ci∑||x−μi||2

  where CiC_iCi is a cluster and μi\mu_iμi its centroid.

8️⃣ Visualize Cluster Distribution

```
plt.figure(figsize=(8, 5))
sns.countplot(x='Cluster', data=df, palette='viridis')
plt.title('Cluster Distribution')
plt.show()
```

✅ Shows how many records belong to each cluster.

📘 Interpretation:

- Balanced clusters → good segmentation.
- One dominant cluster → potential overlap or incorrect `k` value.

9️⃣ Optional: Feature Relationship Visualization

```
if 'SALES' in numeric_df.columns and 'QUANTITYORDERED' in numeric_df.columns:
    plt.figure(figsize=(8, 6))
    sns.scatterplot(
        x=df['SALES'], y=df['QUANTITYORDERED'], hue=df['Cluster'], palette='Set2'
    )
    plt.title('Sales vs Quantity by Cluster')
    plt.show()
```

✅ Plots **Sales vs Quantity Ordered**, color-coded by cluster.

📘 Concept:
This helps interpret business meaning, e.g.:

- Cluster 0 → high sales, large quantity orders (bulk buyers)
- Cluster 1 → medium sales, fewer orders (regular customers)

🔟 Hierarchical Clustering

```
linked = linkage(scaled_data, method='ward')
plt.figure(figsize=(10, 6))
dendrogram(linked, truncate_mode='lastp', p=10, show_leaf_counts=True)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Cluster Size')
plt.ylabel('Distance')
plt.show()
```

✅ Performs **Hierarchical Clustering** and visualizes it as a **dendrogram**.

📘 Concepts:

- **Linkage (Ward method):** merges clusters that minimize variance increase.
- **Dendrogram:** shows how clusters are merged step-by-step.
  You can **cut** the tree at a certain height to form clusters.

▧ Comparison:

Algorithm	Type	Output	Pros
K-Means	Partition-based	Fixed k clusters	Fast for large datasets
Hierarchical	Tree-based	Dendrogram	Doesn't require pre-set k

✓ Completion Message

```
print("\n✓ Clustering complete! Cluster labels added as 'Cluster' column.")
print(df[['Cluster']].head())
```

✓ Indicates successful clustering and shows a preview of cluster assignments.

📋 Concept Summary by Subject

Subject	Concept	Role
Machine Learning	K-Means, Hierarchical clustering	Unsupervised learning
Statistics	Normalization, distance metrics	Ensures fair distance comparison
Data Mining	Pattern recognition	Detects customer segments
Business Analytics	Cluster interpretation	Helps identify high-value customers

---

📈 Business Insights (Practical Interpretation)

Once clustering is done, you can:

- Identify **top-selling segments** (high SALES cluster)
- Detect **low-performing regions** or **products**
- Build **targeted marketing campaigns** per customer cluster
- Optimize **inventory management**

☐ Quick Recap of Key Learnings

Step	Concept	Why It's Important
Scaling	StandardScaler	Ensures equal feature weight
Elbow Method	Optimal k	Prevents over/under clustering
K-Means	Centroid-based	Fast, efficient
Hierarchical	Tree-based	Visually insightful
Visualization	Scatter & dendrogram	Interpret patterns clearly

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score, mean_squared_error

# 1. Load data
df = pd.read_csv("uber.csv")
print("Initial data preview:")
print(df.head())

# 2. Preprocess
df = df.dropna()
df = df[df['fare_amount'] > 0]

# 3. Remove outliers
df = df[df['fare_amount'] < 100]

# 4. Correlation heatmap (numeric columns only)
numeric_df = df.select_dtypes(include=[np.number])
plt.figure(figsize=(8, 5))
sns.heatmap(numeric_df.corr(), annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap (Numeric Features Only)")
plt.show()

# 5. Feature Selection
X = df[['passenger_count', 'pickup_longitude', 'pickup_latitude',
        'dropoff_longitude', 'dropoff_latitude']]
y = df['fare_amount']

# 6. Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 7. Models
lr = LinearRegression()
rf = RandomForestRegressor(random_state=42, n_estimators=100)

lr.fit(X_train, y_train)
rf.fit(X_train, y_train)

# 8. Evaluation
for model, name in zip([lr, rf], ['Linear Regression', 'Random Forest']):
    y_pred = model.predict(X_test)
    r2 = r2_score(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    print(f"\n{name} Performance:")
    print(f"R² Score: {r2:.4f}")
    print(f"RMSE: {rmse:.2f}")
```

PROJECT OVERVIEW

🎯 Goal:

Predict the **fare amount** of Uber rides based on trip details like pickup/dropoff locations and passenger count.

📊 Techniques Used:

- **Linear Regression** → simple interpretable model
- **Random Forest Regression** → non-linear ensemble model
- **Evaluation Metrics:** R² and RMSE

💻 STEP-BY-STEP EXPLANATION

---

1 Load the Data

```
df = pd.read_csv("uber.csv")
print("Initial data preview:")
print(df.head())
```

✅ Reads the Uber dataset and previews the top rows.

📘 Concepts:

- A dataset usually includes:
  - fare_amount: target variable (continuous value)
  - pickup_latitude, pickup_longitude
  - dropoff_latitude, dropoff_longitude
  - passenger_count, datetime, etc.

You always inspect it first to detect missing values, data types, and outliers.

2️⃣ Data Preprocessing

```
df = df.dropna()
df = df[df['fare_amount'] > 0]
```

✅ Removes missing values and unrealistic fare values (e.g., negative fares).

📘 **Concept:**

- **Data cleaning** ensures better model accuracy.
- Dropping NaN values avoids errors during model training.

3️⃣ Remove Outliers

```
df = df[df['fare_amount'] < 100]
```

✅ Filters out unusually high fares (>100), which could be outliers or incorrect entries.

📗 **Why:**
Regression models are sensitive to **outliers**.
Removing them helps improve the generalization of predictions.

4️⃣ Correlation Heatmap

```
numeric_df = df.select_dtypes(include=[np.number])
plt.figure(figsize=(8, 5))
sns.heatmap(numeric_df.corr(), annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap (Numeric Features Only)")
plt.show()
```

✅ Visualizes **feature correlations** among numeric variables.

📘 **Concept:**

- Correlation (r) measures how strongly features move together.
- Range: -1 (negative) to +1 (positive).
- Helps identify redundant features or strong predictors.

📊 **Example Insight:**
If pickup_latitude and dropoff_latitude are strongly correlated, distance may not add much new information.

5⃣ Feature Selection

```
X = df[['passenger_count', 'pickup_longitude', 'pickup_latitude',
        'dropoff_longitude', 'dropoff_latitude']]
y = df['fare_amount']
```

✅ Selects **independent variables (features)** and **dependent variable (target)**.

📘 **Concepts:**

- **Features (X):** Trip parameters that affect fare.
- **Target (y):** Fare amount.
- Feature engineering could also involve:
  - Calculating trip distance
  - Extracting pickup time (rush hour, etc.)

6⃣ Train-Test Split

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

✅ Divides data into:

- **80% training**
- **20% testing**

📘 **Concept:**

- Prevents overfitting by testing on unseen data.
- `random_state` ensures reproducibility.

7⃣ Train Models

```
lr = LinearRegression()
rf = RandomForestRegressor(random_state=42, n_estimators=100)

lr.fit(X_train, y_train)
rf.fit(X_train, y_train)
```

✅ Fits two regression models:

🔹 *Linear Regression*

- Learns a straight-line relationship:

  y^=β0+β1x1+β2x2+...+βnxn\hat{y} = \beta_0 + \beta_1x_1 + \beta_2x_2 + ... + \beta_nx_ny^=β0+β1x1+β2x2+...+βnxn

- Simple, interpretable, but **assumes linearity**.

🔹 *Random Forest Regressor*

- Ensemble of decision trees.
- Handles non-linearity and feature interactions.
- Robust to outliers and noise.

📃 **Concept:**
Random forests can capture **complex geographic effects** (like certain pickup areas having higher fares).

8️⃣ Evaluate Models

```
for model, name in zip([lr, rf], ['Linear Regression', 'Random Forest']):
    y_pred = model.predict(X_test)
    r2 = r2_score(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    print(f"\n{name} Performance:")
    print(f"R² Score: {r2:.4f}")
    print(f"RMSE: {rmse:.2f}")
```

✅ Tests both models and compares performance using **R^2** and **RMSE**.

Metrics Explained

Metric	Meaning	Formula	Ideal
R^2 (Coefficient of Determination)	Measures how much variance in y is explained by model	R2=1−SSresSStotR^2 = 1 - \frac{SS_{res}}{SS_{tot}}R2=1−SStotSSres	Closer to 1
RMSE (Root Mean Squared Error)	Average error magnitude	1n∑(yi−yi^)2\sqrt{\frac{1}{n}\sum(y_i - \hat{y_i})^2}n1∑(yi−yi^)2	Lower is better

📊 **Interpretation Example:**

- Linear Regression → $R^2 = 0.65$, RMSE = 5.8
- Random Forest → $R^2 = 0.85$, RMSE = 3.1
  → **Random Forest performs better**, capturing nonlinear location effects.

📈 BUSINESS INTERPRETATION

Insight	Explanation
Trip Distance vs Fare	Fares likely increase with pickup-dropoff distance.
Passenger Count Effect	Larger groups may slightly increase fare (multi-passenger bookings).
Location Effects	Some areas (like airports) could have higher base fares.
Model Utility	Helps predict future fares, detect anomalies, or optimize pricing.

ML + Data Science Concepts Used

Step	Concept	Field
Preprocessing	Data Cleaning, Outlier Removal	Data Engineering
Correlation	Feature Analysis	Statistics
Linear Regression	Linear Model	Machine Learning
Random Forest	Ensemble Learning	ML (Nonlinear Regression)
R^2, RMSE	Model Evaluation	Model Metrics

🔍 RECOMMENDED NEXT STEPS

You can extend this project by:

1. **Feature Engineering**
   - Compute `trip_distance` using lat/long via the Haversine formula.
   - Extract **hour, day, month** from pickup datetime.
2. **Model Improvement**
   - Use `Gradient Boosting` or `XGBoost` for better accuracy.
3. **Visualization**
   - Plot predicted vs actual fare.
4. **Deployment**
5.
   - Build a Flask web app for fare prediction.