1 — Fibonacci (recursive + non-recursive) — detailed explanation

What the program does (basic)

Computes the nth Fibonacci number using two methods:

- **Recursive**: directly implements the recurrence $F(n) = F(n-1) + F(n-2)$with base cases $F(0) = 0, F(1) = 1$.
- **Iterative (non-recursive)**: computes Fibonacci in a loop maintaining the last two values.

Why two methods

- Recursive is the most direct expression of the mathematical definition — easy to read but inefficient due to repeated subproblems.
- Iterative is linear time and constant extra space — practical for large n.

Recursive implementation — step-by-step

1. Function fibonacci_recursive(n):
    - If n <= 1: return n (covers 0 and 1).
    - Else: return fibonacci_recursive(n-1) + fibonacci_recursive(n-2).
2. Execution trace (example n=5):
    - F(5) calls F(4) and F(3).
    - F(4) calls F(3) and F(2), etc.
    - Many calls compute the same subproblems (F(3) computed multiple times).

Iterative implementation — step-by-step

1. If n <= 1, return n.
2. Initialize a=0 (F(0)), b=1 (F(1)).
3. Loop from 2 to n:
    - a, b = b, a + b — shift the window and compute next Fibonacci.
4. After loop, b holds F(n).

Complexity and reasoning

- Recursive:
    - **Time**: $T(n) = T(n-1) + T(n-2) + O(1)$. Solves to $O(\varphi^n)$where $\varphi \approx 1.618$. Common shorthand: **O(2^n)** (exponential).
    - **Space**: recursion depth is n in worst case → **O(n)** (for call stack).
- Iterative:
    - **Time**: single loop up to n → **O(n)**.

- **Space**: only two variables (a,b) plus constants → **O(1)**.

Practical note

For large n use:

- **Memoization** (top-down DP) or **bottom-up DP** to get O(n) time with O(n) or O(1) space (depending on whether you keep full table).
- Or use matrix exponentiation / fast doubling to compute in $O(\log\ n)$time.

---

2 — Huffman Encoding (Greedy) — detailed explanation

2. Huffman Encoding (Greedy Strategy)

Function Name	Purpose / Use in Program
build_huffman_tree(freqs)	**Builds the Huffman tree from character frequencies using a min-heap. Repeatedly merges two least frequent nodes (Greedy approach).**
build_codes(node, prefix, codebook)	**Traverses the Huffman tree to assign binary codes (0/1) to each character. Generates variable-length prefix codes.**
huffman_encode(s)	**Orchestrates the encoding: builds frequency table, constructs tree, and encodes input string into binary form.**
huffman_decode(encoded_str, root)	**Decodes the binary string by traversing the Huffman tree from root to leaves. Reconstructs the original message.**

What the program does (basic)

Given a set of characters and their frequencies, it builds a Huffman tree to assign variable-length binary codes so that more frequent characters get shorter codes — optimal prefix code that minimizes total weighted code length.

Key ideas (greedy insight)

- Build a min-heap (priority queue) keyed by frequency.
- Repeatedly extract the two nodes with smallest frequencies, combine them into a new node whose frequency is the sum, and push back to the heap.
- This greedy pairing of smallest frequencies is provably optimal (Huffman's algorithm).

Data structures used

- **Node** object (has char, freq, left, right). Internal nodes have char = None.

- **Min-heap** using Python heapq — supports O(log n) push/pop.

Step-by-step algorithm

1. Create leaf Node for each character with its frequency and push into min-heap.
2. While heap size > 1:
    - Pop left = heapq.heappop(heap) (smallest freq).
    - Pop right = heapq.heappop(heap) (next smallest).
    - Create newNode with freq = left.freq + right.freq, left and right children set.
    - Push newNode back into heap.
3. Remaining node in heap is root of Huffman tree.
4. Generate codes by traversing tree:
    - Left edge adds "0" to code, right edge adds "1".
    - At each leaf, assign accumulated code string to that character.

Example

Input chars=['a','b','c','d','e','f'] with freq=[5,9,12,13,16,45] yields codes like f:0, c:100... (exact codes depend on tie breaks).

Complexity

- Building heap initially: O(n).
- Each of the (n−1) merges requires two heappop and one heappush — each O(log n) → total **O(n log n)** time.
- Traversal to generate codes: O(n) (visiting each node once).
- **Space**: O(n) for nodes and heap.

Implementation notes & pitfalls

- If many characters have equal frequencies, tie-breaking may change specific code words but not total length.
- The algorithm yields a prefix code: no code is a prefix of another — so decoding is unambiguous.
- For large alphabets or streaming data, building frequency counts may be the dominant cost.

(Reference implementation style: repository examples for DAA exercises). GitHub

---

3 — 0/1 Knapsack (Dynamic Programming) — detailed explanation

Problem statement (basic)

Given n items each with weight w[i] and value v[i], and capacity W, pick a subset so total weight ≤ W and value is maximized. This is the classic 0/1 knapsack (each item either taken or not).

DP idea (bottom-up)

Define dp[i][w] = maximum value achievable with first i items and capacity w.

- Base: dp[0][*] = 0 and dp[*][0] = 0.
- Transition: for item i (1-based):
    - If weights[i-1] <= w: dp[i][w] = max(values[i-1] + dp[i-1][w - weights[i-1]], dp[i-1][w])
    - Else: dp[i][w] = dp[i-1][w]

Step-by-step

1. Create 2D table dp with dimensions (n+1) x (W+1) initialized to 0.
2. Fill rows i=1..n and columns w=1..W using the transition above.
3. Answer is dp[n][W].
4. To recover which items are chosen, backtrack from dp[n][W]:
    - If dp[i][w] != dp[i-1][w], item i was included; set w -= weights[i-1], i -= 1.
    - Else i -= 1 (item not included).

Complexity

- Time: loop over i and w → **O(nW)**.
- Space: storing table → **O(nW)**.
- You can reduce space to **O(W)** by using a 1D array and iterating capacities in reverse for each item (to avoid overwriting dependencies), but you lose easy backtracking unless you store extra info.

Implementation notes

- DP is pseudo-polynomial: polynomial in W not in the numeric size of W. If W is large, DP can be expensive.
- If item values are small but weights large, alternative DP by value is possible (minimize weight for each value sum).
- Branch-and-bound is useful when W or n too large for DP; it prunes using upper bounds (e.g., fractional knapsack relaxation).

(Repository contains similar lab code style for knapsack). GitHub

---

4 — N-Queens with first queen fixed (Backtracking) — detailed explanation

Problem (basic)

Place n queens on an n×n chessboard so that no two queens attack each other (no same row, column, or diagonal). In your problem you **start with the first queen already placed** at some position; the algorithm must place remaining queens.

Backtracking idea

Place queens row by row (or column by column). For row r:

1. Try each column c:
    - If position (r, c) is **safe** (no other queen in same column or diagonals), place queen.
    - Recurse to next row r+1.
    - If recursion returns with solution(s), continue collecting them (or stop after one solution).
    - Backtrack: remove queen from (r, c) and try next column.

When the first queen is preplaced at row 0 and column k, you start recursion at row 1, and ensure position constraints respect the preplaced queen.

Safety checks

- Column check: iterate previous rows i=0..r-1, ensure board[i][c] != 1.
- Upper-left diagonal: decrement i and j (row, col) together.
- Upper-right diagonal: decrement i, increment j until bounds.

Optimizations:

- Use boolean arrays for cols, diag1, diag2 to check safety in O(1) rather than scanning rows:
    - cols[c] = True if column c occupied.
    - diag1[r+c] for one diagonal direction, diag2[r-c + n-1] for the other.
    - This reduces safety check from O(n) to O(1) and speeds up search.

Complexity

- Time: backtracking explores permutations of queens; worst-case complexity is **O(n!)** (very rough). Exact counts are smaller due to pruning but still factorial growth.
- Space: $O(n^2)$ for board (or O(n) if using compressed representations) + recursion stack O(n).

Implementation notes

- If you need **all** solutions, let recursion continue after a solution is found; if you need one, return early.
- Fixing the first queen reduces the search space (symmetry broken), but depending on position might make the remainder harder/easier.

---

5 — QuickSort (Deterministic & Randomized) — detailed explanation + analysis

QuickSort basics (divide & conquer)

- Choose a pivot element.
- Partition array into elements ≤ pivot (left) and > pivot (right) and place pivot in its final sorted position.
- Recursively sort left and right subarrays.

Deterministic pivot choice

- Common choices: last element, first element, median-of-three.
- Deterministic implementation often uses pivot = arr[high] (last element) and Lomuto partition scheme or Hoare partition scheme.

Deterministic partition (Lomuto)

1. Pivot = arr[high].
2. i = low - 1.
3. For j from low to high-1:
    - If arr[j] <= pivot: i += 1; swap arr[i], arr[j].
4. After loop swap arr[i+1] and arr[high] (place pivot).
5. Return pivot index i+1.

Randomized QuickSort

- Pick pivot uniformly at random from low..high and swap it with arr[high], then run same partition.
- Randomizing eliminates worst-case input patterns (like already sorted arrays) with very high probability.

Complexity analysis

- Best case: pivot splits array evenly each time → recurrence $T(n) = 2T(n/2) + O(n)$→ **O(n log n)**.
- Average case (random pivot): **O(n log n)** (expected).
- Worst case (poor pivot every time: e.g., smallest or largest) for deterministic last-element pivot on sorted input: $T(n) = T(n-1) + O(n)$→ **O(n^2)**.
- For randomized pivot: worst case still O(n^2) in absolute terms, but probability of encountering worst case in a run is extremely small → expected **O(n log n)**.
- Space: recursion stack depth is expected O(log n); worst-case O(n) → **O(log n)** average, **O(n)** worst.

Practical implementation tips

- Use randomized pivot if you cannot control input distribution.
- Use cutoff to insertion sort for small subarrays (e.g., length < 10) for performance improvements.
- Use tail recursion elimination or iterative loop to limit recursion depth on one side.

Cross-algorithm comments, testing & sample outputs

Testing tips

- For each algorithm include small, known testcases and also randomly generated tests.
- For Huffman: build a short sentence frequency map, encode and then decode to verify.
- For Knapsack: test small capacities where you can verify optimal subset by hand.
- For N-Queens: verify number of solutions for small n (n=4 has 2 solutions, n=8 has 92).
- For QuickSort: test with sorted, reverse sorted, duplicates, and random arrays to observe behavior of deterministic vs randomized.

Where the GitHub repository helps

- The repository you shared (SPPU DAA Python folder) contains student lab implementations and is useful for seeing example input/output patterns and simple file organization used in labs. Use it as a coding style reference (function names, basic I/O). GitHub

6. Predict the Price of an Uber Ride using Linear Regression and Random Forest Regression

Objective

To predict the fare (price) of an Uber ride based on features like distance, pickup time, drop-off time, and location.

Dataset Example Attributes

- Pickup latitude and longitude
- Drop-off latitude and longitude
- Time of day (hour, weekday/weekend)
- Distance (calculated from coordinates)
- Traffic or surge pricing factor
- Fare amount (target variable)

---

a) Linear Regression

Concept:
Linear Regression is a **supervised learning algorithm** used for predicting a continuous value.

It assumes a **linear relationship** between independent variables (X) and dependent variable (Y).

Mathematical Model:

$$Y = b_0 + b_1X_1 + b_2X_2 + \ldots + b_nX_n + \varepsilon$$

Where:

- Y: predicted fare
- X_i: feature variables (distance, time, etc.)
- b_i: coefficients
- ε: error term

Advantages:

- Easy to interpret
- Works well for linearly correlated data

Limitations:

- Fails for nonlinear relationships
- Sensitive to outliers

---

b) Random Forest Regression

Concept:
Random Forest is an **ensemble learning technique** combining multiple **decision trees** to improve prediction accuracy and reduce overfitting.

Working:

- Random subsets of data and features are used to build multiple trees.
- Final prediction is the **average of all decision trees' outputs.**

Advantages:

- Handles nonlinear data
- Less overfitting than single decision trees
- Can handle large feature sets

Evaluation Metrics:

- **Mean Absolute Error (MAE)**
- **Mean Squared Error (MSE)**
- **R^2 (Coefficient of Determination)**

Comparison:

- Linear Regression: interpretable but less accurate for nonlinear data.
- Random Forest: more accurate and robust but less interpretable.

---

7. Email Spam Detection using KNN and SVM

Objective

To classify emails as **Spam (1)** or **Not Spam (0)** using machine learning.

Dataset Attributes

- Word frequency counts
- Presence of special symbols or links
- Email length, sender reputation, etc.
- Target: Spam (1) / Not Spam (0)

---

a) K-Nearest Neighbors (KNN)

Concept:
KNN is a **non-parametric**, **instance-based learning** algorithm used for classification.

Working:

1. Choose the value of **K** (number of neighbors).
2. Calculate the **Euclidean distance** between the test email and all training emails.
3. Select K nearest neighbors.
4. Assign the majority class among the neighbors.

Advantages:

- Simple and easy to implement
- No training phase

Limitations:

- Computationally expensive for large datasets
- Sensitive to irrelevant features and scaling

---

b) Support Vector Machine (SVM)

Concept:
SVM is a **supervised classification** algorithm that finds an optimal **hyperplane** separating two classes.

Mathematical Model:

$$w \cdot x + b = 0$$

The goal is to **maximize the margin** between classes.

Advantages:

- Works well for high-dimensional data (like text)
- Effective for non-linear data using **kernel tricks**

Kernels: Linear, Polynomial, Radial Basis Function (RBF)

Performance Analysis:

- **Confusion Matrix:** True Positives, False Positives, etc.
- **Accuracy:** (TP+TN)/(TP+TN+FP+FN)
- **Precision & Recall**
- **F1 Score**

Comparison:

- KNN: Simple but slower prediction.
- SVM: More accurate and robust for text classification.

---

8. Bank Customer Churn Prediction using Neural Network

Objective

To predict if a customer will **leave (churn)** or **stay** with the bank in the next 6 months.

Dataset Attributes

- Customer ID, Age, Gender
- Credit Score
- Account Balance
- Number of Products
- Tenure (years with bank)
- IsActiveMember
- Exited (Target variable)

Concept: Neural Networks

A **neural network** is inspired by the human brain and consists of interconnected layers of neurons.

Structure:

1. **Input Layer:** Takes features (e.g., age, balance).
2. **Hidden Layers:** Perform weighted computations and apply activation functions.
3. **Output Layer:** Gives probability of churn (0 = No, 1 = Yes).

Activation Functions:

- ReLU (Hidden layers)
- Sigmoid (Output for binary classification)

Training Process:

- Forward propagation
- Compute error using loss function (Binary Cross Entropy)
- Backpropagation using gradient descent

Evaluation:

- Accuracy
- Precision, Recall, F1-score
- ROC-AUC Curve

---

9. Diabetes Prediction using KNN

Objective

To predict whether a patient has diabetes based on medical diagnostic features.

Dataset: diabetes.csv

Common attributes:

- Pregnancies
- Glucose level
- Blood Pressure
- Insulin
- BMI
- Age
- Outcome (1 = Diabetic, 0 = Not Diabetic)

KNN Working

- Compute distances between the test data and training samples.

- Select the **K nearest neighbors**.
- Assign the class based on the majority vote.

Metrics:

1. **Confusion Matrix:**
   Shows TP, TN, FP, FN.
2. **Accuracy:** Correct predictions / Total predictions
3. **Error Rate:** 1 - Accuracy
4. **Precision:** TP / (TP + FP)

Advantages:

- Simple, effective for small datasets
  Disadvantages:
- Slow for large datasets
- Requires feature scaling

---

10. Customer Segmentation using K-Means or Hierarchical Clustering

Objective

To group customers or sales data into clusters with similar characteristics.

Dataset: sales_data_sample.csv

Features like:

- Customer ID
- Region
- Sales Amount
- Product Category
- Quantity

---

a) K-Means Clustering

Concept:
An **unsupervised learning algorithm** that groups data into **K clusters** based on feature similarity.

Steps:

1. Choose number of clusters **K**.
2. Initialize **K centroids** randomly.

3. Assign each data point to the nearest centroid.
4. Recalculate centroids until convergence.

Elbow Method:
Used to determine optimal **K** by plotting **Within-Cluster Sum of Squares (WCSS)** vs. K.
The "elbow" point shows the optimal K.

---

b) Hierarchical Clustering

Concept:
Builds a hierarchy of clusters using **agglomerative (bottom-up)** or **divisive (top-down)** approaches.

Visualization: Dendrogram (shows merging of clusters)

Distance Metrics: Euclidean, Manhattan
Linkage Methods: Single, Complete, Average

---

11. Installation of MetaMask and Studying Ether Transactions

Objective

To understand blockchain transactions, wallets, and gas fees using **MetaMask**.

MetaMask Overview

MetaMask is a **browser-based Ethereum wallet** used for managing crypto assets and interacting with decentralized applications (DApps).

Steps:

1. Install MetaMask as a browser extension.
2. Create or import a wallet (12-word seed phrase).
3. Connect to Ethereum **test networks** (e.g., Sepolia, Goerli).
4. Obtain test Ether using **faucets**.
5. Observe **transaction details** like:
    - Transaction Hash
    - Gas Used
    - Gas Price
    - Total Transaction Fee

Gas Fee:
Fee paid to miners for validating transactions.

$$\text{Gas Fee} = \text{Gas Used} \times \text{Gas Price}$$

---

12. Smart Contract for Bank Account Operations

Objective

To develop and deploy a **smart contract** simulating a bank account with Deposit, Withdraw, and Balance operations.

Concept:

A **smart contract** is a self-executing agreement written in Solidity, deployed on the Ethereum blockchain.

Main Functions:

- **Deposit():** Add Ether to account.
- **Withdraw():** Withdraw specified amount (only if sufficient balance).
- **getBalance():** Returns current balance.

Key Concepts:

- **msg.sender:** Address of caller.
- **msg.value:** Amount of Ether sent.
- **require():** Condition check for validity.
- **Mapping:** Store balances for each address.

Testing:
Deploy on Remix IDE using a test network (e.g., Sepolia).
Check transaction details and gas consumption.

---

13. Student Data Smart Contract using Structures, Arrays & Fallback

Objective

To create a Solidity contract that stores student data and manages Ether transactions using fallback functions.

Concepts:

Structures:
Used to define custom data types.
Example:
struct Student { uint id; string name; uint marks; }

Arrays:
To store multiple student records dynamically.

Fallback Function:
A special function executed when:

- Contract receives Ether without function call.
- Nonexistent function is called.

Purpose:

- Accept Ether
- Log or handle unexpected calls

Gas and Transaction Fee Observation:
After deployment, note:

- **Gas Limit**
- **Gas Used by Transaction**
- **Gas Price**
- **Total Fee (in ETH)**

Understanding Gas:
Gas measures computational effort — more complex functions consume more gas.

✅ **Summary Table**

Exp No	Topic	Techniques Used	Key Concepts	Type
6	Uber Ride Price Prediction	Linear Regression, Random Forest	Regression, MAE, R^2	Supervised
7	Email Spam Detection	KNN, SVM	Classification, Confusion Matrix	Supervised
8	Customer Churn	Neural Network	Activation, Backpropagation	Supervised
9	Diabetes Prediction	KNN	Confusion Matrix, Accuracy	Supervised
10	Customer Segmentation	K-Means, Hierarchical	Clustering, Elbow Method	Unsupervised
11	MetaMask Study	Ethereum Wallet	Gas Fees, Wallets	Blockchain
12	Bank Smart Contract	Solidity	Deposit, Withdraw, Balance	Blockchain
13	Student Contract	Solidity	Struct, Array, Fallback	Blockchain

⚙️ **PRACTICAL QUESTIONS & ANSWERS ON SMART CONTRACTS**

Q1. What is a Smart Contract in Blockchain?

Answer:
A **Smart Contract** is a **self-executing program** stored on the blockchain that automatically enforces the terms of an agreement when predefined conditions are met.
It eliminates intermediaries and ensures transparency and security.

Example use:
Transferring funds, managing digital assets, or automating transactions like deposits and withdrawals.

Q2. Which programming language is used for writing Ethereum Smart Contracts?

Answer:
Solidity — a high-level, contract-oriented language designed specifically for writing smart contracts that run on the **Ethereum Virtual Machine (EVM)**.

Q3. What is the function of msg.sender, msg.value, and msg.data in Solidity?

Answer:

- **msg.sender** → Address of the account that called the function.
- **msg.value** → Amount of Ether (in wei) sent with the transaction.
- **msg.data** → Complete calldata (information) sent with the transaction.

Example:

```
function deposit() public payable {
   balances[msg.sender] += msg.value;
}
```

Here, msg.sender identifies who sent the Ether and msg.value specifies how much Ether was sent.

Q4. What is the difference between a constructor and a fallback function in Solidity?

Answer:

Feature	Constructor	Fallback
Purpose	Initializes the contract	Handles Ether or unknown calls
Execution	Runs **only once** during deployment	Runs **whenever Ether is sent** or unknown function is called
Declaration	constructor() { }	fallback() external payable { }

Feature	Constructor	Fallback
Payment	Not payable by default	Must be marked payable to accept Ether

---

Q5. Write the purpose of the require() and assert() functions in Solidity.

Answer:

- **require(condition, message)** → Used for input validation and pre-condition checks. If the condition fails, it **reverts** the transaction and refunds remaining gas.
- **assert(condition)** → Used for internal error checking. Failing assert consumes all remaining gas and indicates a **bug** in code logic.

Example:

```
require(balance[msg.sender] >= amount, "Insufficient funds");
```

Q6. What is Gas in Ethereum?

Answer:
Gas is a unit that measures the computational effort required to execute operations on the Ethereum network.
Each instruction (storage, function call, loop) consumes a fixed amount of gas.

Formula:

$$\text{Total Fee (in ETH)} = \text{Gas Used} \times \text{Gas Price}$$

Q7. Explain the use of the payable keyword in Solidity.

Answer:
The payable keyword allows a function or address to **receive Ether**.
Without payable, a function cannot accept any Ether attached to a transaction.

Example:

```
function deposit() public payable {
  balance[msg.sender] += msg.value;
}
```

Q8. What is the difference between public and private visibility in Solidity functions or variables?

Answer:

Visibility	Accessible From	Description
Public	Inside & Outside contract	Visible to everyone, including external contracts
Private	Only inside same contract	Not accessible by derived contracts

---

Q9. What is the purpose of a Mapping in Solidity?

Answer:
A **mapping** is a key-value data structure used to store and retrieve data efficiently in smart contracts.

Syntax:

```
mapping(address => uint) public balances;
```

It links each user's **address** to their **balance**.
For example, you can retrieve balance by balances[msg.sender];

Q10. Explain what happens when Ether is sent to a contract that doesn't have a payable function.

Answer:
If Ether is sent to a contract that **does not have a payable fallback or receive() function**, the transaction will **fail** and the Ether will be **reverted** back to the sender.

Q11. What is the purpose of the receive() function in Solidity?

Answer:
The receive() function is a special function used to accept Ether sent directly to a contract **without data**.
If Ether is sent with data or no matching function, then the **fallback()** function is executed instead.

Syntax:

```
receive() external payable {
   // Accept Ether
}
```

Q12. What are events in Solidity and why are they used?

Answer:
Events are used for **logging information** on the blockchain.
They help external applications (like web apps or wallets) listen for and respond to specific contract activities.

Example:

event Deposit(address indexed user, uint amount);

function deposit() public payable {

  emit Deposit(msg.sender, msg.value);

}

External interfaces can track Deposit events without scanning the entire blockchain.

Q13. What is MetaMask and why is it used?

Answer:
MetaMask is a crypto wallet and gateway to Ethereum-based applications.
It allows users to:

- Store and manage private keys
- Send and receive Ether
- Interact with DApps and smart contracts on blockchain
- Connect to **test networks** (e.g., Sepolia, Goerli) for learning and testing

---

Q14. How can you observe Gas Fees while deploying a contract using Remix and MetaMask?

Answer:

1. Connect MetaMask to a **test network**.
2. Compile and deploy the contract via **Remix IDE**.
3. MetaMask pops up with transaction details.
4. Observe:
    - **Gas Limit**
    - **Gas Used**
    - **Gas Price (in Gwei)**
    - **Total Fee (ETH equivalent)**

You can view exact gas details on **Etherscan** by pasting the transaction hash.

Q15. Explain what happens when you withdraw Ether using a smart contract function.

Answer:
When the **withdraw()** function is executed:

1. It checks if the sender's balance ≥ withdrawal amount (using require()).
2. The amount is subtracted from the sender's balance.
3. Ether is transferred to the sender using:
4. payable(msg.sender).transfer(amount);
5. The blockchain updates the transaction record, deducting gas fees.

Q16. What are the two main types of Ethereum test networks used for smart contract deployment?

Answer:

1. **Goerli Testnet** – Previously used for testing contracts.
2. **Sepolia Testnet** – Current active testnet for contract deployment and testing Ether transactions.

Both networks use **test Ether**, which can be obtained from **faucets** for free.

Q17. What is a fallback function used for in Solidity?

Answer:
A **fallback function** executes when:

- Ether is sent **with data** or
- A **non-existent function** is called.

It ensures the contract can handle unexpected transactions safely.

Syntax:

```
fallback() external payable {
  // Handle Ether or invalid calls
}
```

Q18. Why are Smart Contracts immutable once deployed?

Answer:
Once deployed to the Ethereum blockchain, a smart contract's bytecode is **permanently stored** on the ledger.
This ensures:

- **Security** (no unauthorized modifications)

- **Transparency** (code visible to all)
  But also means errors cannot be fixed — developers must deploy a new version if needed.

Q19. What are the main sections of a Solidity contract?

Answer:

1. **Pragma directive** – Specifies compiler version (e.g., pragma solidity ^0.8.0;)
2. **Contract declaration** – Defines the contract name
3. **State variables** – Store data (balances, user info)
4. **Constructor** – Initializes state when deployed
5. **Functions** – Define logic (deposit, withdraw, etc.)
6. **Events** – For transaction logging

Q20. What are the advantages of Smart Contracts?

Answer:

- **Automation:** Executes automatically without intermediaries
- **Transparency:** Code is visible on blockchain
- **Security:** Tamper-proof and immutable
- **Cost-effective:** Reduces need for middlemen
- **Speed:** Faster execution than manual systems

- # 🎒 3. 0–1 Knapsack Problem (Dynamic Programming / Branch and Bound)

Function Name	Purpose / Use in Program
`knapsack_dp(weights, values, W)`	Builds a **DP table** to find the maximum profit possible without exceeding capacity `W`. Each cell `dp[i][w]` stores the best value using the first `i` items and capacity `w`. Time complexity = O(nW).
`reconstruct_items(dp, weights, values, W)`	Traces back through the DP table to determine which items were included in the optimal solution. Useful for displaying selected items.
`knapsack_branch_and_bound(items, W)` *(optional)*	Implements **branch and bound** approach for large problems. Uses bounding (like fractional knapsack upper bound) to prune suboptimal branches. Demonstrates optimization beyond pure DP.

- ---
- ## 👑 4. N-Queens Problem (Backtracking with First Queen Fixed)

Function Name	Purpose / Use in Program
`is_safe(board, row, col)`	Checks if placing a queen at `(row, col)` is safe — i.e., no other queen attacks it vertically, horizontally, or diagonally. Core pruning logic.
`solve_nqueens_with_first(n, first_row, first_col)`	Initializes the board with the first queen fixed, then recursively places remaining queens using **backtracking** until a valid arrangement is found.
`print_board(board)`	Displays the final N×N matrix showing the positions of queens (1 for queen, 0 for empty cell). Helps visualize the solution.

- ---
- ## ⚡ 5. Quick Sort (Deterministic and Randomized)

Function Name	Purpose / Use in Program
`partition(arr, low, high, pivot_index)`	Core function that rearranges array elements so that those less than pivot come before it, and greater elements come after. Returns pivot's final position.
`quicksort_deterministic(arr, low, high)`	Implements **Quick Sort** using a fixed or median pivot (deterministic). Demonstrates typical divide-and-conquer recursion.
`quicksort_randomized(arr, low, high)`	Uses **random pivot selection** to avoid worst-case scenarios. Shows how randomization improves expected runtime performance.

- ---
- ## 🚖 6. Uber Ride Price Prediction (Linear Regression & Random Forest)

Function Name	Purpose / Use in Program
`load_data(file_path)`	Loads Uber trip data (pickup/drop coordinates, distance, duration, etc.) into a DataFrame.
`preprocess_data(df)`	Handles missing values, feature scaling, categorical encoding, and feature engineering (e.g., distance from lat-long).
`train_linear_regression(X_train, y_train)`	Trains a **Linear Regression** model to learn a linear relationship between ride features and fare amount.
`train_random_forest(X_train, y_train)`	Trains a **Random Forest Regressor** — captures nonlinear relationships between features.
`evaluate_model(model, X_test, y_test)`	Computes metrics like RMSE, MAE, and R^2 to compare model performance. Demonstrates how

Function Name	Purpose / Use in Program
	Random Forest usually outperforms simple linear models.

- ---

- ## 📧 7. Email Spam Detection (KNN & SVM Classification)

Function Name	Purpose / Use in Program
`load_email_data(file_path)`	Loads dataset containing labeled emails (spam / not spam).
`text_preprocessing(emails)`	Cleans and tokenizes email text — removing stopwords, punctuation, and converting to lowercase.
`vectorize_text(data)`	Converts processed text into numerical vectors using **TF-IDF** or **CountVectorizer** for model input.
`train_knn_classifier(X_train, y_train)`	Trains **K-Nearest Neighbors** model — classifies email based on similarity to nearest neighbors.
`train_svm_classifier(X_train, y_train)`	Trains **Support Vector Machine** classifier — finds best separating hyperplane between spam and ham classes.
`evaluate_classifiers(models, X_test, y_test)`	Calculates accuracy, precision, recall, and F1-score to compare both classifiers' performance.

- ---

- ## 🧠 8. Bank Customer Churn Prediction (Neural Network Classifier)

Function Name	Purpose / Use in Program
`load_customer_data(file_path)`	Reads the bank's customer churn dataset with features like age, balance, tenure, etc.
`preprocess_data(df)`	Encodes categorical variables, scales numerical data, and splits dataset into training/testing.
`build_neural_network(input_dim)`	Defines the **Artificial Neural Network (ANN)** architecture using layers (Dense, Dropout, Activation).
`train_model(model, X_train, y_train, epochs, batch_size)`	Trains the network using backpropagation and optimizers like Adam or RMSProp.
`evaluate_model(model, X_test, y_test)`	Computes performance metrics such as accuracy, confusion matrix, and ROC curve for churn prediction.

- ---

- 💉 **9. Diabetes Prediction using K-Nearest Neighbors (KNN)**

Function Name	Purpose / Use in Program
`load_diabetes_data(file_path)`	Loads diabetes dataset (features like glucose, BMI, insulin, etc.).
`normalize_features(X)`	Performs **feature scaling** so all features contribute equally in distance computation.
`train_knn_model(X_train, y_train, k)`	Trains a **KNN classifier** using the value of `k` (number of neighbors).
`predict_knn(model, X_test)`	Predicts diabetes outcome (positive/negative) for test samples.
`evaluate_performance(y_test, y_pred)`	Computes **confusion matrix**, **accuracy**, **error rate**, and **precision**. Essential for performance analysis.

- ---

- 📊 **10. Sales Data Clustering (K-Means / Hierarchical Clustering)**

Function Name	Purpose / Use in Program
`load_sales_data(file_path)`	Loads sales or customer dataset (features like product category, revenue, quantity, etc.).
`preprocess_data(df)`	Cleans and scales numeric features before clustering.
`apply_kmeans(X, k)`	Performs **K-Means Clustering** — partitions data into k clusters based on feature similarity.
`determine_optimal_k(X)`	Uses **Elbow Method** to identify best value of `k` by plotting inertia vs. k.
`apply_hierarchical_clustering(X)`	Applies **Agglomerative Hierarchical Clustering** and visualizes using dendrogram.
`visualize_clusters(X, labels)`	Plots final clusters to interpret sales patterns or customer segments.