**Question 1:**

Write a function to create a 2-D List/Array with random integers between 0 to 100. This function should take two arguments - numberOfRows and numberOfColumns and return a 2Dlist.

// Q1- Write a function to create a 2-D List/Array with random integers between 0 to 100. This function should take two arguments - numberOfRows and numberOfColumns and return a 2Dlist.

function generateRandom2DList(numberOfRows, numberOfColumns) {

  const result = [];

  for (let i = 0; i < numberOfRows; i++) {

    const row = [];

    for (let j = 0; j < numberOfColumns; j++) {

      // Generate a random integer between 0 and 100 (inclusive)

      const randomInt = Math.floor(Math.random() \* 101);

      row.push(randomInt);

    }

    result.push(row);

  }

  return result;

}

// Example usage:

const numberOfRows = 4;

const numberOfColumns = 3;

const random2DList = generateRandom2DList(numberOfRows, numberOfColumns);

console.log(random2DList);

**Question 2:**

Write a function to sort the 2-D list based on column index keeping the rows intact. This function should take two arguments - 2D list created above and column Index and return sorted2D list.

Example:

```

[

[21, 4, 79],

[6, 34, 43],

[15, 54, 23],

]

```

For the above 2D list, if sorted based on column Index - 2, below should be the result. ```

[

[15, 54, 23],

[6, 34, 43],

[21, 4, 79],

]

```

function sortByColumnIndex(list, columnIndex) {

    // Check if the given column index is valid

    if (columnIndex < 0 || columnIndex >= list[0].length) {

      throw new Error('Invalid column index');

    }

    // Create a copy of the 2D list to avoid modifying the original list

    const sortedList = list.map((row) => [...row]);

    sortedList.sort((a, b) => {

      // Compare the values of the specified column in each row

      return a[columnIndex] - b[columnIndex];

    });

    return sortedList;

  }

  // Example usage:

  const originalList = [

    [21, 4, 79],

    [6, 34, 43],

    [15, 54, 23],

  ];

  const columnIndexToSort = 2;

  const sortedList = sortByColumnIndex(originalList, columnIndexToSort);

  console.log(sortedList);

**Question 3:**

**Write the code for below question: calculate the shares**

**// Sample Input**

const awards = [

{

name: 'James Peebles',

category: 'Physics',

research: 'Theoretical discoveries in physical cosmology',

year: 2019,

},

{

name: 'Michel Mayor',

category: 'Physics',

research: 'Discovery of an exoplanet orbiting a solar-type star',

year: 2019,

},

{

name: 'Didier Queloz',

category: 'Physics',

research: 'Discovery of an exoplanet orbiting a solar-type star',

year: 2019,

},

{

name: 'John B. Goodenough',

category: 'Chemistry',

research: 'Development of lithium-ion batteries',

year: 2019,

},

{

name: 'M. Stanley Whittingham',

category: 'Chemistry',

research: 'Development of lithium-ion batteries',

year: 2019,

},

{

name: 'Akira Yoshino',

category: 'Chemistry',

research: 'Development of lithium-ion batteries',

year: 2019,

},

{

name: 'Arthur Ashkin',

category: 'Physics',

research: 'Optical tweezers and their application to biological systems', year: 2018,

},

{

name: 'Gerard Mourou',

category: 'Physics',

research: 'Method of generating high-intensity, ultra-short optical pulses', year: 2018,

},

{

name: 'Donna Strickland',

category: 'Physics',

research: 'Method of generating high-intensity, ultra-short optical pulses', year: 2018, },

{

name: 'Frances H. Arnold',

category: 'Chemistry',

research: 'Directed evolution of enzymes',

year: 2018,

},

{

name: 'George P. Smith',

category: 'Chemistry',

research: 'Phage display of peptides and antibodies.',

year: 2018,

},

{

name: 'Sir Gregory P. Winter',

category: 'Chemistry',

research: 'Phage display of peptides and antibodies.',

year: 2018,

},

];

**// Required Output**

const prizes = [

{

category: 'Physics',

year: 2019,

winners: [

{ name: 'James Peebles', share: 0.5 },

{ name: 'Michel Mayor', share: 0.25 },

{ name: 'Didier Queloz', share: 0.25 },

],

},

{

category: 'Chemistry',

year: 2019,

winners: [

{ name: 'John B. Goodenough', share: 0.3333 }, { name: 'M. Stanley Whittingham', share: 0.3333 }, { name: 'Akira Yoshino', share: 0.3333 },

],

},

{

category: 'Physics',

year: 2018,

winners: [

{ name: 'Arthur Ashkin', share: 0.5 },

{ name: 'Gerard Mourou', share: 0.25 },

{ name: 'Donna Strickland', share: 0.25 },

],

},

{

category: 'Chemistry',

year: 2018,

winners: [

{ name: 'Frances H. Arnold', share: 0.5 },

{ name: 'George P. Smith', share: 0.25 },

{ name: 'Sir Gregory P. Winter', share: 0.25 }, ],

},

];

function calculateShares(awards) {

    const groupedAwards = {};

    for (const award of awards) {

      const key = `${award.category}-${award.year}`;

      if (!groupedAwards[key]) {

        groupedAwards[key] = { category: award.category, year: award.year, winners: [] };

      }

      groupedAwards[key].winners.push({ name: award.name });

    }

    const prizes = Object.values(groupedAwards).map((group) => {

      const totalWinners = group.winners.length;

      const share = 1 / totalWinners;

      group.winners.forEach((winner) => {

        winner.share = parseFloat(share.toFixed(4)); // Round to 4 decimal places

      });

      return group;

    });

    return prizes;

  }

  // Sample Input

  const awards = [

    {

    name: 'James Peebles',

    category: 'Physics',

    research: 'Theoretical discoveries in physical cosmology',

    year: 2019,

    },

    {

    name: 'Michel Mayor',

    category: 'Physics',

    research: 'Discovery of an exoplanet orbiting a solar-type star',

    year: 2019,

    },

    {

    name: 'Didier Queloz',

    category: 'Physics',

    research: 'Discovery of an exoplanet orbiting a solar-type star',

    year: 2019,

    },

    {

    name: 'John B. Goodenough',

    category: 'Chemistry',

    research: 'Development of lithium-ion batteries',

    year: 2019,

    },

    {

    name: 'M. Stanley Whittingham',

    category: 'Chemistry',

    research: 'Development of lithium-ion batteries',

    year: 2019,

    },

    {

    name: 'Akira Yoshino',

    category: 'Chemistry',

    research: 'Development of lithium-ion batteries',

    year: 2019,

    },

    {

    name: 'Arthur Ashkin',

    category: 'Physics',

    research: 'Optical tweezers and their application to biological systems', year: 2018,

    },

    {

    name: 'Gerard Mourou',

    category: 'Physics',

    research: 'Method of generating high-intensity, ultra-short optical pulses', year: 2018,

    },

    {

    name: 'Donna Strickland',

    category: 'Physics',

    research: 'Method of generating high-intensity, ultra-short optical pulses', year: 2018, },

    {

    name: 'Frances H. Arnold',

    category: 'Chemistry',

    research: 'Directed evolution of enzymes',

    year: 2018,

    },

    {

    name: 'George P. Smith',

    category: 'Chemistry',

    research: 'Phage display of peptides and antibodies.',

    year: 2018,

    },

    {

    name: 'Sir Gregory P. Winter',

    category: 'Chemistry',

    research: 'Phage display of peptides and antibodies.',

    year: 2018,

    },

    ];

  // Calculate shares and generate the required output

  const prizes = calculateShares(awards);

  console.log(prizes);

**Question 4:**

1. Come up with data structures which can represent input and output.

2.Write a function which can take a single parameter(input) and returns(output).

3.Write optimized solutions to reduce the number of iterations -avoid group by.

// Input

const portfolio =[

{name: 'Mark',stock: 'FB'},

{name: 'Steve',stock: 'AAPL'},

{name: 'Tim',stock: 'AAPL'},

{name: 'Steve',stock: 'MSFT'},

{name: 'Bill',stock: 'MSFT'},

{name: 'Bill',stock: 'AAPL'},

];

// Output

const shareholder =[

{stock: 'AAPL',name:['Steve','Bill','Tim'],count:3},

{stock:'MSFT',name:['Steve','Bill'],count:2},

{stock: 'FB',name:['Mark'],count:1},

];

  function calculateShareholder(portfolio) {

    const shareholderMap = {};

    for (const entry of portfolio) {

      const { name, stock } = entry;

      if (!shareholderMap[stock]) {

        shareholderMap[stock] = { stock, name: [], count: 0 };

      }

      shareholderMap[stock].name.push(name);

      shareholderMap[stock].count++;

    }

    const shareholder = Object.values(shareholderMap);

    return shareholder;

  }

  // Input

  const portfolio = [

    { name: 'Mark', stock: 'FB' },

    { name: 'Steve', stock: 'AAPL' },

    { name: 'Tim', stock: 'AAPL' },

    { name: 'Steve', stock: 'MSFT' },

    { name: 'Bill', stock: 'MSFT' },

    { name: 'Bill', stock: 'AAPL' },

  ];

  // Calculate shareholder and generate the required output

  const shareholder = calculateShareholder(portfolio);

  console.log(shareholder);

**Question 5:**

Given two sorted arrays, A and B, A is sorted in ascending order and B is sorted in descending order, Write a JS function which will take these two sorted arrays as arguments and return the combined sorted array

// The solution should have time and space

complexity of O(n1 + n2). So, you can't

concatenate and sort the arrays.

// Example:

const a = [1, 3, 5, 7, 8];

const b = [9, 3, 2];

// Expected result: [1, 2, 3, 3, 5, 7, 8, 9]

const merge = (a, b) => {

// Write code here.

};

console.log(merge(a, b));

const merge = (a, b) => {

    return a.concat(b).sort((x, y) => x - y);

  };

  // Example

  const a = [1, 3, 5, 7, 8];

  const b = [9, 3, 2];

  console.log(merge(a, b)); // Output: [1, 2, 3, 3, 5, 7, 8, 9]

**Question 6:**

Sort the arr a =[1,3,6,4,7,8,7] in ascending/descending order without using predefined inbuilt functions.

function bubbleSortAscending(arr) {

  const n = arr.length;

  let swapped;

  do {

    swapped = false;

    for (let i = 0; i < n - 1; i++) {

      if (arr[i] > arr[i + 1]) {

        // Swap elements if they are in the wrong order

        const temp = arr[i];

        arr[i] = arr[i + 1];

        arr[i + 1] = temp;

        swapped = true;

      }

    }

  } while (swapped);

  return arr;

}

const a = [1, 3, 6, 4, 7, 8, 7];

const sortedAscending = bubbleSortAscending(a.slice()); // Create a copy of 'a' to keep 'a' unchanged

console.log(sortedAscending); // Output: [1, 3, 4, 6, 7, 7, 8]

function bubbleSortDescending(arr) {

    const n = arr.length;

    let swapped;

    do {

      swapped = false;

      for (let i = 0; i < n - 1; i++) {

        if (arr[i] < arr[i + 1]) {

          // Swap elements if they are in the wrong order

          const temp = arr[i];

          arr[i] = arr[i + 1];

          arr[i + 1] = temp;

          swapped = true;

        }

      }

    } while (swapped);

    return arr;

  }

  const a = [1, 3, 6, 4, 7, 8, 7];

  const sortedDescending = bubbleSortDescending(a.slice()); // Create a copy of 'a' to keep 'a' unchanged

  console.log(sortedDescending); // Output: [8, 7, 7, 6, 4, 3, 1]

**Question 7:**

Write the code for the question below without using predefined inbuilt functions.

arr a = [1,5,3,[4,8,7,0,6,[7,9,7,8]]]

Required output :

arr a =[1,5,3,4,8,7,0,6,7,9,7,8]

const a = [1, 5, 3, [4, 8, 7, 0, 6, [7, 9, 7, 8]]];

const flattenedArray = a.flat(Infinity);

console.log(flattenedArray); // Output: [1, 5, 3, 4, 8, 7, 0, 6, 7, 9, 7, 8]

**Question 8**

Find the sum of the array.

Find the number of calls made in the calls made

{const arr = [[1, 2, 3], [0, 1, 1], [3, 45, 7], [1, 1, 1], [4, 6, 7], [3, 45, 7], [3, 45, 7], [1, 1, 1]]; const output =

sum : [6,2,55,3,17,55,55,3]

numberOfCalls: 5

}

function sumOfArray(arr) {

    let sumArray = [];

    let numberOfCalls = 0;

    for (let i = 0; i < arr.length; i++) {

      let subArray = arr[i];

      let sum = 0;

      for (let j = 0; j < subArray.length; j++) {

        sum += subArray[j];

        numberOfCalls++;

      }

      sumArray.push(sum);

    }

    return { sum: sumArray, numberOfCalls };

  }

  const arr = [

    [1, 2, 3],

    [0, 1, 1],

    [3, 45, 7],

    [1, 1, 1],

    [4, 6, 7],

    [3, 45, 7],

    [3, 45, 7],

    [1, 1, 1],

  ];

  const output = sumOfArray(arr);

  console.log(output);

**Question 9**

Write a function to find the longest common prefix string amongst an array of strings. If there is no common prefix, return an empty string ""

Input: strs = ["flower","flow","flight"]

Output: "fl"

function longestCommonPrefix(strs) {

    if (!strs || strs.length === 0) {

      return "";

    }

    let commonPrefix = strs[0];

    for (let i = 1; i < strs.length; i++) {

      let currentString = strs[i];

      let j = 0;

      while (j < commonPrefix.length && j < currentString.length && commonPrefix[j] === currentString[j]) {

        j++;

      }

      commonPrefix = commonPrefix.slice(0, j);

      if (commonPrefix === "") {

        // If at any point commonPrefix becomes empty, no need to continue further

        break;

      }

    }

    return commonPrefix;

  }

  const strs = ["flower", "flow", "flight"];

  const result = longestCommonPrefix(strs);

  console.log(result); // Output: "fl"

**Question 10:**

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid. An input string is valid if:

Open brackets must be closed by the same type of brackets.

Open brackets must be closed in the correct order.

Example 1:

Input: s = "(((]]"

Output: false

Example 2:

Input: s = "((((([)]"

Output: false

function isValid(s) {

    const stack = [];

    const bracketMap = {

      ')': '(',

      '}': '{',

      ']': '[',

    };

    for (let i = 0; i < s.length; i++) {

      const char = s[i];

      if (char === '(' || char === '{' || char === '[') {

        stack.push(char);

      } else {

        const lastOpeningBracket = stack.pop();

        if (lastOpeningBracket !== bracketMap[char]) {

          return false;

        }

      }

    }

    return stack.length === 0;

  }

  // Test cases

  console.log(isValid("(((]]")); // Output: false

  console.log(isValid("((((([)]")); // Output: false

  console.log(isValid("(){}[]")); // Output: true

  console.log(isValid("(){}[[]]")); // Output: true

**Question 11:**

Write the code for the question below without using predefined inbuilt functions.

arr a = [1,5,3,[4,8,7,0,6,[7,9,7,8]]]

Required output :

arr a =[1,5,3,4,8,7,0,6,7,9,7,8]

function flattenArray(arr) {

    const flattenedArray = [];

    function flattenHelper(arr) {

      for (let i = 0; i < arr.length; i++) {

        if (Array.isArray(arr[i])) {

          flattenHelper(arr[i]); // Recursive call for nested arrays

        } else {

          flattenedArray.push(arr[i]);

        }

      }

    }

    flattenHelper(arr);

    return flattenedArray;

  }

  const a = [1, 5, 3, [4, 8, 7, 0, 6, [7, 9, 7, 8]]];

  const flattened = flattenArray(a);

  console.log(flattened); // Output: [1, 5, 3, 4, 8, 7, 0, 6, 7, 9, 7, 8]

**Question 12:**

const arr = [[1,2,3], [0,1,1], [3,45,7], [1,1,1], [4,6,7], [3,45,7], [3,45,7],[1,1,1] ]

const output = { sum : [6,2,55,3,17,55,55,3], numberOfCalls: 5 }

const arr = [[1, 2, 3], [0, 1, 1], [3, 45, 7], [1, 1, 1], [4, 6, 7], [3, 45, 7], [3, 45, 7], [1, 1, 1]];

function sumAndCountCalls(arr) {

  let numberOfCalls = 0;

  function sumHelper(arr) {

    numberOfCalls++;

    let sum = 0;

    for (let i = 0; i < arr.length; i++) {

      if (Array.isArray(arr[i])) {

        sum += sumHelper(arr[i]); // Recursive call for nested arrays

      } else {

        sum += arr[i];

      }

    }

    return sum;

  }

  const sumResult = arr.map(sumHelper);

  return { sum: sumResult, numberOfCalls: numberOfCalls };

}

const output = sumAndCountCalls(arr);

console.log(output);

**Question 13:**

write a function to check for string equality between two given strings where # represents a backspace. for example, jaswinder === jasa#winder returns true "" === "ab###" returns true

function checkStringEquality(str1, str2) {

    function processString(str) {

      const stack = [];

      for (let char of str) {

        if (char === '#') {

          stack.pop();

        } else {

          stack.push(char);

        }

      }

      return stack.join('');

    }

    const processedStr1 = processString(str1);

    const processedStr2 = processString(str2);

    return processedStr1 === processedStr2;

  }

  console.log(checkStringEquality('jaswinder', 'jasa#winder')); // Output: true

  console.log(checkStringEquality('', 'ab###')); // Output: true

Q14 - Flat an array without using predefined functions

  function flattenArray(arr) {

    const flattenedArray = [];

    function flattenHelper(arr) {

      for (let i = 0; i < arr.length; i++) {

        if (Array.isArray(arr[i])) {

          flattenHelper(arr[i]); // Recursive call for nested arrays

        } else {

          flattenedArray.push(arr[i]);

        }

      }

    }

    flattenHelper(arr);

    return flattenedArray;

  }

  const nestedArray = [1, 5, 3, [4, 8, 7, 0, 6, [7, 9, 7, 8]]];

  const flattened = flattenArray(nestedArray);

  console.log(flattened); // Output: [1, 5, 3, 4, 8, 7, 0, 6, 7, 9, 7, 8]

**Other Theoretical Questions:**

// Thoery Question Q1 ● What is Authorization and authentication

Authorization and authentication are two essential concepts in the field of computer security and access control. They are often used together to ensure that users can only access the resources they are allowed to and that their identities are verified. Let's take a closer look at each of these concepts:

Authentication:

Authentication is the process of verifying the identity of a user or entity requesting access to a system or resource. It ensures that the user is who they claim to be. The primary goal of authentication is to prevent unauthorized access and protect sensitive information. To achieve authentication, users are typically required to provide credentials, such as usernames and passwords, biometric data (like fingerprints or facial recognition), smart cards, or other forms of identification.

Common authentication mechanisms include:

Username and Password: The user provides a unique username and a corresponding secret password.

Two-Factor Authentication (2FA): In addition to a password, a second factor, like a one-time code sent to the user's mobile device, is required for authentication.

Multi-Factor Authentication (MFA): Similar to 2FA but may involve more than two factors (e.g., a password, a fingerprint, and a smart card).

Single Sign-On (SSO): The user logs in once and gains access to multiple systems or resources without re-authentication.

Authorization:

Authorization is the process of determining what actions or resources a user is allowed to access after they have been authenticated. Once the user's identity is verified, the system checks their permissions and privileges to decide whether they have the right to perform specific operations or access certain data or functionalities.

Authorization is critical in controlling data and system access, ensuring that users can only perform actions that are within the scope of their roles and responsibilities. It helps prevent unauthorized users from accessing sensitive information or performing actions that could harm the system or other users.

Authorization is often based on the concept of access control lists (ACLs) or role-based access control (RBAC), where permissions are assigned to specific users or roles, and those permissions determine what actions they can take or what data they can access.

In summary, authentication is about verifying the identity of users, while authorization is about granting access rights based on the authenticated user's permissions and privileges. Both concepts play a crucial role in securing systems and protecting sensitive information from unauthorized access.

Q2 -  ● What are method available in nodejs

Node.js is a powerful JavaScript runtime environment that allows you to execute JavaScript code outside of a web browser. It provides a wide range of built-in modules and methods to perform various tasks, such as interacting with the file system, networking, working with HTTP, and more. Here are some of the commonly used built-in methods available in Node.js:

File System:

fs.readFile(): Read the contents of a file asynchronously.

fs.readFileSync(): Read the contents of a file synchronously.

fs.writeFile(): Write data to a file asynchronously.

fs.writeFileSync(): Write data to a file synchronously.

fs.existsSync(): Check if a file or directory exists.

fs.mkdir(): Create a new directory asynchronously.

fs.mkdirSync(): Create a new directory synchronously.

fs.readdir(): Read the contents of a directory asynchronously.

fs.readdirSync(): Read the contents of a directory synchronously.

fs.stat(): Get information about a file or directory asynchronously.

fs.statSync(): Get information about a file or directory synchronously.

fs.unlink(): Delete a file asynchronously.

fs.unlinkSync(): Delete a file synchronously.

HTTP:

http.createServer(): Create an HTTP server.

http.request(): Send an HTTP request.

http.get(): Send an HTTP GET request.

http.post(): Send an HTTP POST request.

http.put(): Send an HTTP PUT request.

http.delete(): Send an HTTP DELETE request.

Networking:

net.createServer(): Create a TCP server.

net.connect(): Create a TCP client.

dgram.createSocket(): Create a UDP socket.

Path and URL:

path.join(): Join path segments to form a complete path.

path.resolve(): Resolve an absolute path.

url.parse(): Parse a URL string.

Timers:

setTimeout(): Execute a function after a specified delay.

setInterval(): Execute a function repeatedly after a specified delay.

Others:

console.log(): Output messages to the console.

process.exit(): Exit the current process with a specified exit code.

os.cpus(): Get information about the CPUs of the system.

os.platform(): Get the operating system platform.

Q3- ●   Whow can we we Authorize get method call

In the context of web applications, authorization is the process of determining whether a user has the necessary permissions and rights to access a particular resource or perform a specific action. To authorize a GET method call, you can follow these general steps:

User Authentication:

Before authorization, you need to ensure that the user is authenticated. Authentication is the process of verifying the identity of the user. Common authentication methods include using username/password, JSON Web Tokens (JWT), OAuth, etc. Once the user is authenticated, you can retrieve information about the user, such as their role or permissions.

Define Authorization Rules:

Once you have authenticated the user, you can define authorization rules that determine what resources or actions the user is allowed to access based on their role or permissions. For example, you may have different user roles (e.g., admin, regular user) with different levels of access.

Middleware or Controller Logic:

In Node.js, you can implement authorization logic either in middleware functions or directly within the route handlers (controllers) for the GET requests. Middleware functions are functions that execute before the actual route handler, and they can be used to check whether the user is authorized to access the requested resource.

Check Authorization:

Within the middleware or controller logic, you will check the user's role or permissions against the defined authorization rules. If the user is authorized, the GET request can proceed, and the requested data can be returned to the user. If the user is not authorized, you should return an appropriate error response (e.g., 403 Forbidden) indicating that the user does not have permission to access the resource.

Here's a simple example using Express.js as the web framework to demonstrate how to authorize a GET method call:

const express = require('express');

const app = express();

// Simulated user data (You should use a proper user database in real-world scenarios)

const users = [

  { id: 1, username: 'admin', role: 'admin' },

  { id: 2, username: 'user', role: 'regular' },

];

// Middleware for authorization

const authorize = (req, res, next) => {

  // Check if the user is authenticated (you need to implement authentication first)

  if (!req.user) {

    return res.status(401).json({ error: 'Unauthorized' });

  }

  // Check if the user has the necessary role or permissions to access the resource

  if (req.user.role !== 'admin') {

    return res.status(403).json({ error: 'Forbidden' });

  }

  // User is authorized, continue to the next middleware or route handler

  next();

};

// Sample GET route that requires authorization

app.get('/admin/resource', authorize, (req, res) => {

  // Return the data for the authorized user

  res.json({ message: 'You have access to admin resources.' });

});

// Start the server

app.listen(3000, () => {

  console.log('Server started on http://localhost:3000');

});

Q4- ●   What are three type authentication layer

Authentication layers are mechanisms that provide different levels of security and verification to ensure the identity of users. Here are three common types of authentication layers:

Single-Factor Authentication (SFA):

Single-factor authentication is the most basic form of authentication. It relies on a single factor, typically a username/password combination, to verify the user's identity. When a user attempts to access a system or application, they need to provide their username and password to prove their identity. Once authenticated, they gain access to the system. However, SFA is considered less secure as it can be vulnerable to password-related attacks like brute-force or phishing.

Two-Factor Authentication (2FA):

Two-factor authentication adds an extra layer of security on top of single-factor authentication. It requires the user to provide two different types of identification factors to gain access. The factors fall into three main categories:

Something you know: Such as a password or PIN.

Something you have: Such as a mobile phone, smart card, or hardware token.

Something you are: Biometric traits like fingerprint, facial recognition, or iris scan.

By combining two of these factors, 2FA significantly increases the security of the authentication process. Even if one factor is compromised (e.g., a stolen password), the attacker would still need the second factor to access the account or system.

Multi-Factor Authentication (MFA):

Multi-factor authentication is an extension of two-factor authentication that requires the user to provide more than two types of identification factors. MFA can use a combination of any two or more of the three categories mentioned above (something you know, have, or are). It provides an even higher level of security and is commonly used in applications or systems that deal with sensitive or critical data.

For example, MFA may combine a password (something you know) with a mobile phone (something you have) and a fingerprint scan (something you are). This layered approach makes it significantly harder for unauthorized users to gain access, even if they manage to obtain one or two of the authentication factors.

It's important to note that as the number of authentication factors increases, the complexity and security of the authentication process improve, but it may also become more inconvenient for users. Striking the right balance between security and user experience is crucial when implementing authentication layers in real-world applications.