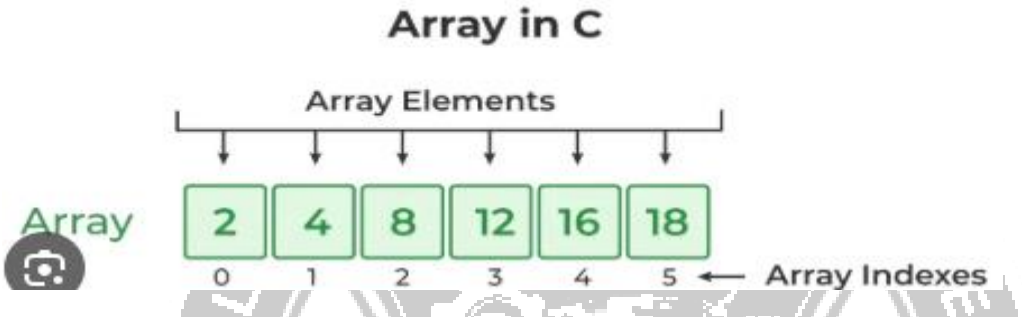
# 1.2. ARRAYS, STRINGS, STRUCTURES, REAL-WORLD USE CASES (E.G., TEXT PROCESSING, SENSORS)

An array is a foundational data structure that stores a collection of items of the same type at contiguous memory locations. It acts like a single variable containing multiple values, with each item assigned a specific numerical position (an "index") starting at




**Array Introduction**

An array is a collection of items of the same variable type that are stored at contiguous memory locations. It is one of the most popular and simple data structures used in programming.

Basic terminologies of Array

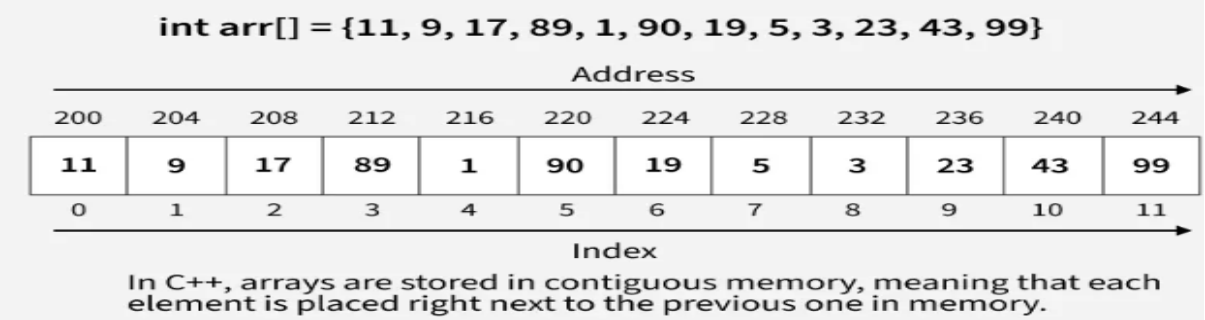
- **Array Element:** Elements are items stored in an array.
- **Array Index:** Elements are accessed by their indexes. Indexes in most of the programming languages start from 0.

Memory representation of Array

In an array, all the elements or their references are stored in contiguous memory locations. This allows for efficient access and manipulation of elements.

Declaration of Array

Arrays can be declared in various ways in different languages. For better illustration, below are some language-specific array declarations:




```
// This array will store integer type element
int arr[5];
// This array will store char type element
char arr[10];
// This array will store float type element
float arr[20];
```

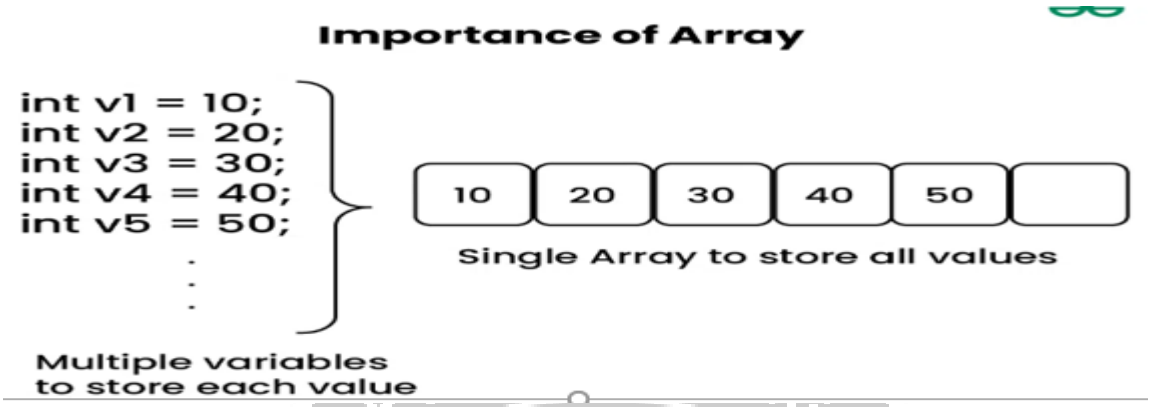
Initialization of Array

Arrays can be initialized in different ways in different languages. Below are some language-specific array initialization:

```
int arr[] = { 1, 2, 3, 4, 5 };
char arr[5] = { 'a', 'b', 'c', 'd', 'e' };
float arr[10] = { 1.4, 2.0, 24, 5.0, 0.0 };
```

Why do we Need Arrays?

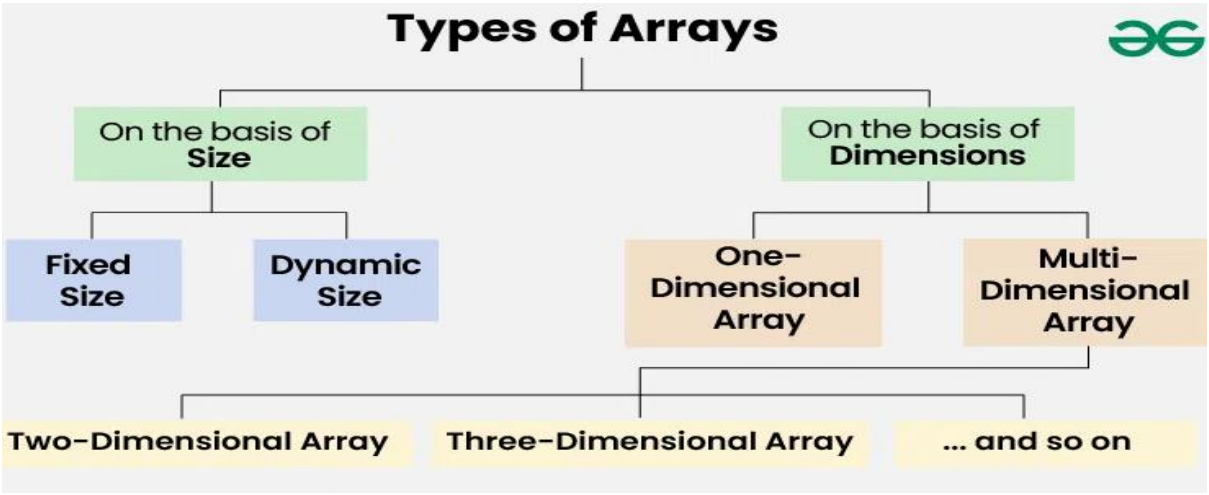
Assume there is a class of five students and if we have to keep records of their marks in examination then, we can do this by declaring five variables individual and keeping track of records but what if the number of students becomes very large, it would be challenging to manipulate and maintain the data. So we use an array of students.



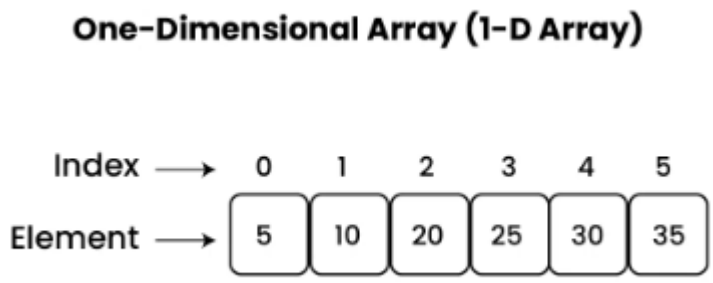

Types of Arrays

Arrays can be classified in two ways:

- On the basis of Size
- On the basis of Dimensions

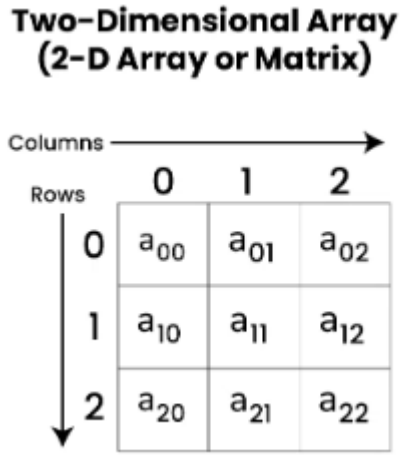



**1. One-dimensional Array(1-D Array):** You can imagine a 1d array as a row, where elements are stored one after another.



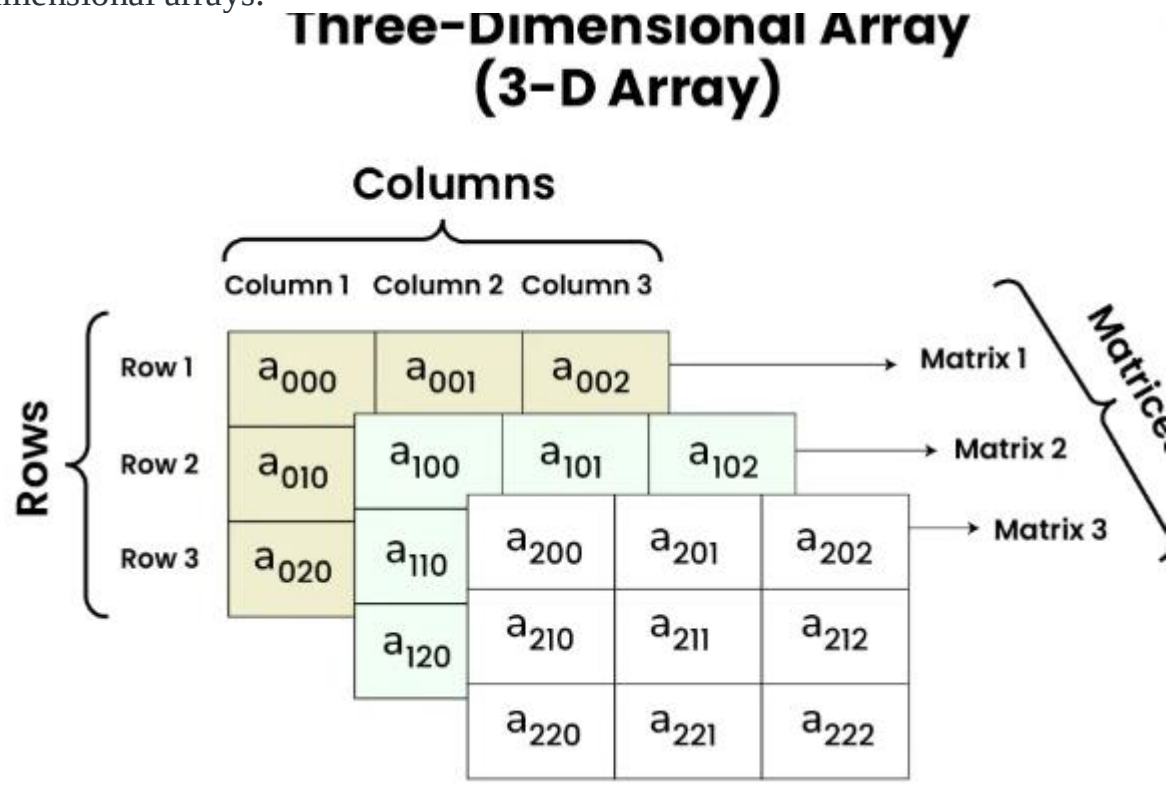
**2. Multi-dimensional Array:** A multi-dimensional array is an array with more than one dimension. We can use multidimensional array to store complex data in the form of tables, etc. We can have 2-D arrays, 3-D arrays, 4-D arrays and so on.

- **Two-Dimensional Array(2-D Array or Matrix):** 2-D Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.




**T**

**Three-Dimensional Array(3-D Array):** A 3-D Multidimensional array contains three dimensions, so it can be considered an array of two-dimensional arrays.




## STRINGS IN C

**A** string in C **is a one-dimensional array of char type, with the last character in the array being a "null character" represented by '\0'. Thus, a string in C can be defined as a null-terminated sequence of char type values.**

## Creating a String in C

A string in C is an array of characters terminated by a null character '\0'.

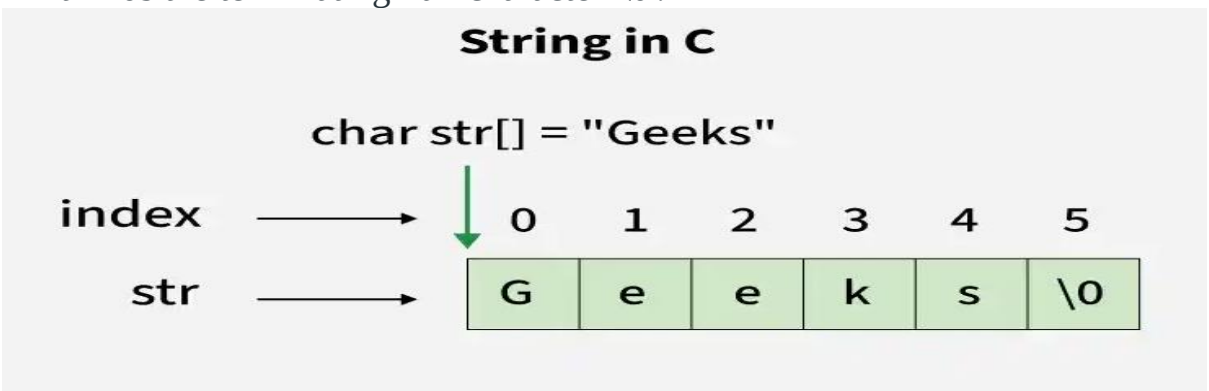
- The null character '\0' marks the end of the string.
- C does not have a built-in string data type.
- Strings are implemented using arrays of char.

```
#include <stdio.h>

int main()

// declaring and initializing a string

   char str[] = "Geeks"

 // printing the string

   printf("The string is: %s\n", str)

 return 0;

}
```

**Output**

```
The string is: Geeks
```

- **char str[] = "Geeks";** This line declares a character array str and initializes it with the string "Geeks". Internally, this creates an array like: { 'G', 'e', 'e', 'k', 's', '\0'}
  The null character '\0' is automatically added at the end to terminate the string.
- **printf("The string is: %s\n", str);** %s is the format specifier used to print a string. printf starts at the memory location of str and prints each character until it finds the terminating null character '\0'.



Accessing Characters

We can access any character of the string by providing the index (position) of the character, like in array.

```
#include <stdio.h>

int main() {

 char str[] = "Geeks";

    // Access first character

    // of string

    printf("%c", str[0]);

    return 0;

}
```

**Output**

```
G
```

UPDATE

We can change individual characters of a string using their index: str[0] = 'h'. Strings can also be updated using standard library functions like strcpy() to replace the entire string. Ensure the new string fits within the allocated array size to avoid memory issues.

```
#include <stdio.h>

int main() {

    char str[] = "Geeks"

    // Update the first

    // character of string

    str[0] = 'R';

    printf("%c", str[0]);

    return 0;

}
```

**Output**

```
R
```

Str

STRING LENGTH

To find the length of a string in C, you need to iterate through each character until you reach the null terminator '\0', which marks the end of the string. This process is handled efficiently by the **strlen()** function from the C standard library.

```
#include <stdio.h>

int main() {
    char str[] = "Geeks";
   printf("%d", strlen(str));
    return 0;
}
```

**Output**

```
5
```

In this example, **strlen()** returns the length of the string "**Geeks**", which is **5**, excluding the null character.

## STRUCTURES IN C

A **structure** in C is a derived or user-defined data type. We use the keyword **struct** to define a custom data type that groups together the elements of different types. The difference between an array and a structure is that an **array** is a homogenous collection of similar types, whereas a structure can have elements of different types stored adjacently and identified by a name.

We are often required to work with values of different **data types** having certain relationships among them. For a **book** is described by its **title** (string), **author** (string), **price** (double), **number of pages** (integer), etc. Instead of using four different variables, these values can be stored in a single **struct** variable. example,

### Declare (Create) a Structure

You can create (declare) a structure by using the **"struct"** keyword followed by the structure_tag (structure name) and declare all of the members of the structure inside the curly braces along with their data types.

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member.

## Syntax of Structure Declaration

The format (syntax) to declare a structure is as follows −

```
struct [structure tag]{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as "int i;" or "float f;" or any other valid variable definition.

At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is **optional**.

## Example

In the following example we are declaring a structure for Book to store the details of a Book −

```
struct book{
    char  title[50];
    char  author[50];
    double price;
    int   pages;
} book1;
```

Here, we declared the structure variable **book1** at the end of the structure definition. However, you can do it separately in a different statement.

## Structure Variable Declaration

To access and manipulate the members of the structure, you need to declare its variable first. To declare a structure variable, write the structure name along with the **"struct"** keyword followed by the name of the structure variable. This structure variable will be used to access and manipulate the structure members.

### Example

The following statement demonstrates how to declare (create) a structure variable

```
struct book book1;
```

Usually, a structure is declared before the first function is defined in the program, after the **include** statements. That way, the derived type can be used for declaring its variable inside any function.

## Structure Initialization

The **initialization** of a struct variable is done by placing the value of each element inside curly brackets.

### Example

The following statement demonstrates the initialization of structure

```
struct book book1 = {"Learn C", "Dennis Ritchie", 675.50, 325};
```

## Accessing the Structure Members

To access the members of a structure, first, you need to declare a structure variable and then use the **dot (.) operator** along with the structure variable.

### Example 1

The four elements of the struct variable book1 are accessed with the **dot (.) operator**. Hence, "book1.title" refers to the title element, "book1.author" is the author name, "book1.price" is the price, "book1.pages" is the fourth element (number of pages).

Take a look at the following example −

```c
#include <stdio.h>

struct book{
    char title[10];
    char author[20];
    double price;
    int pages;
};

int main(){
    struct book book1 = {"Learn C", "Dennis Ritchie", 675.50, 325};

    printf("Title:  %s \n", book1.title);
    printf("Author: %s \n", book1.author);
    printf("Price:  %lf\n", book1.price);
    printf("Pages:  %d \n", book1.pages);
    printf("Size of book struct: %d", sizeof(struct book));
    return 0;
```

### *Output*

Run the code and check its output −

Title:  Learn C
Author: Dennis Ritchie
Price:  675.500000
Pages:  325
Size of book struct: 48

REAL-WORLD USE CASES (E.G., TEXT PROCESSING, SENSORS),

**Text processing**

Text processing is the automated creation, manipulation, or analysis of electronic text by computers. It converts raw, unstructured text into structured, standardized formats that machines can understand

**1. Text Processing Applications**

Text processing involves analyzing, organizing, searching, and understanding textual data.

What is Text Processing?

Text processing refers to the manipulation and analysis of text data using algorithms.

### Common Operations

- Searching words
- Spell checking
- Text classification
- Language translation
- Sentiment analysis

## REAL-WORLD APPLICATIONS

### A. Search Engines

When users search for information, algorithms process billions of documents to find relevant results.

#### *Algorithms Used*

- String Matching
- Ranking Algorithms
- Indexing Algorithms
- Hashing

#### *Example*

Searching:

"Best laptop under 50000"

The system:

1. Matches keywords.
2. Searches indexed pages.
3. Ranks pages.
4. Displays results.

## B. Spell Checkers

Used in:

- Word processors
- Email systems
- Mobile keyboards

### *Algorithms Used*

- Edit Distance
- Levenshtein Distance
- Dictionary Lookup

### *Example*

Input:

Recieve

Output:

Receive

The algorithm finds the closest valid word.

C. Autocomplete Systems

Used in:

- Search bars
- Mobile keyboards
- E-commerce websites

### *Algorithms Used*

- Trie Data Structure
- Prefix Trees

### *Example*

Typing:

Comp

Suggestions:

Computer
Compiler
Company

D. Machine Translation

Used for converting one language into another.

*Applications*

- Website translation
- International communication
- Language learning

*Algorithms Used*

- Neural Networks
- Sequence Models
- Deep Learning

Example:

English → Tamil

E. Sentiment Analysis

Determines whether text is:

- Positive
- Negative
- Neutral

*Applications*

- Product reviews
- Social media monitoring
- Customer feedback analysis

Example:

"The product is amazing."

Output:

Positive Sentiment

## 2. Sensor-Based Applications

Sensors collect data from the environment.

Examples:

- Temperature sensors
- Motion sensors
- Pressure sensors
- Humidity sensors
- GPS sensors

Algorithms process sensor data to make intelligent decisions.

### A. Smart Homes

#### Sensors Used

- Motion sensors
- Light sensors
- Temperature sensors

#### Algorithms Used

- Rule-based algorithms
- Machine Learning

#### Example

Motion detected:

IF motion detected
THEN turn ON lights

Benefits:

- Energy saving
- Automation
- Convenience

### B. Automatic Doors

Used in:

- Shopping malls
- Airports
- Hospitals

### Sensors Used

Infrared Sensors

### Working

1. Detect approaching person.
2. Trigger opening mechanism.
3. Close after passage.

### Algorithms

- Event Detection
- Signal Processing

C. Smart Thermostats

Used to control room temperature automatically.

### Sensors

- Temperature sensors
- Occupancy sensors

### Algorithms

- Predictive Models
- Control Algorithms

Example:

Temperature > 28°C→ Turn ON AC

## 3. Healthcare Applications

Algorithms play a crucial role in modern healthcare.

A. Patient Monitoring Systems

Monitor:

- Heart rate
- Blood pressure
- Oxygen levels

### Sensors Used

- ECG sensors
- Pulse oximeters

### Algorithms

- Signal Processing
- Pattern Recognition

Example:

Detect abnormal heartbeat and alert doctors.

B. Disease Prediction

Algorithms analyze medical records.

### Used For

- Diabetes prediction
- Cancer detection
- Heart disease risk assessment

### Algorithms

- Decision Trees
- Neural Networks
- Logistic Regression

C. Medical Image Processing

Used in:

- MRI scans
- CT scans
- X-rays

### Algorithms

- Image Segmentation
- Edge Detection
- Deep Learning

Example:

Detect tumors from MRI images.

**4. Transportation Systems**

Algorithms improve transportation efficiency.

### A. GPS Navigation

### Algorithms

- Dijkstra's Algorithm
- A* Algorithm

### Tasks

- Shortest route
- Fastest route
- Traffic avoidance

Example:

Finding fastest route from home to office.

### B. Traffic Management

### Sensors

- Cameras
- Vehicle detectors

### Algorithms

- Traffic Flow Analysis
- Optimization Algorithms

Benefits:

- Reduced congestion

- Improved traffic flow

C. Ride-Sharing Services

Tasks:

- Match drivers and riders
- Calculate fares
- Optimize routes

### Algorithms

- Matching Algorithms
- Route Optimization

**5. Industrial Automation**

Factories use algorithms to automate production.

A. Robotics

Robots perform:

- Welding
- Painting
- Packaging

### Algorithms

- Path Planning
- Computer Vision
- Motion Control

B. Predictive Maintenance

Machines are monitored continuously.

### Sensors

- Vibration sensors
- Temperature sensors

### Algorithms

- Anomaly Detection

- Machine Learning

Benefits:

- Prevent breakdowns
- Reduce maintenance costs

**6. Internet of Things (IoT)**

IoT devices communicate through sensors and networks.

### Examples

### Smart Agriculture

Sensors monitor:

- Soil moisture
- Temperature
- Humidity

### Algorithm Action

IF soil moisture < threshold
THEN start irrigation

Benefits:

- Water conservation
- Increased crop yield

### Smart Cities

Sensors monitor:

- Traffic
- Pollution
- Energy consumption

Algorithms optimize city operations.

**7. Financial Systems**

A. Fraud Detection

Banks analyze transactions.

### Algorithms

- Machine Learning
- Pattern Recognition

Example:

Normal purchase: ₹500
Suspicious purchase: ₹2,00,000 abroad

System flags transaction.

### B. Stock Market Analysis

Algorithms analyze:

- Historical prices
- Market trends

Applications:

- Trading decisions
- Risk assessment

## 8. Social Media Applications

### A. Content Recommendation

Platforms recommend:

- Videos
- Posts
- Friends

### Algorithms

- Collaborative Filtering
- Deep Learning

Example:

Watching technology videos leads to more technology recommendations.

### B. Spam Detection

Algorithms identify:

- Fake accounts
- Spam messages
- Harmful content

## 9. Cybersecurity Applications

### A. Intrusion Detection Systems

Monitor network traffic.

### Algorithms

- Pattern Matching
- Anomaly Detection

Detect:

- Unauthorized access
- Cyber attacks

### B. Encryption

Protects sensitive data.

### Algorithms

- AES
- RSA
- SHA

Applications:

- Online banking
- Secure messaging
- E-commerce

## 10. E-Commerce Applications

### A. Product Recommendation Systems

Suggest products based on:

- Browsing history
- Purchase history
- Similar users

### Algorithms

- Collaborative Filtering
- Recommendation Engines

### B. Inventory Management

Algorithms help:

- Predict demand
- Manage stock levels
- Reduce waste

**Common Algorithms and Their Real-World Use Cases**

| **Algorithm** | **Real-World Use** |
|---|---|
| Linear Search | Contact search |
| Binary Search | Database lookup |
| Bubble Sort | Small datasets |
| Merge Sort | Large database sorting |
| Quick Sort | Programming libraries |
| BFS | Social networks |
| DFS | Maze solving |
| Dijkstra | GPS navigation |
| A* | Route planning |
| Dynamic Programming | Resource optimization |
| Greedy Algorithms | Scheduling |
| Neural Networks | AI applications |

| **Algorithm** | **Real-World Use** |
|---|---|
| K-Means | Customer segmentation |
| Decision Trees | Medical diagnosis |
| Hashing | Password storage |
| Trie | Autocomplete systems |