1. Breadth First Search (BFS) Implementation

BFS is a graph traversal algorithm that explores all the nodes at the present depth level before moving on to nodes at the next depth level. It uses a queue to manage the nodes to visit.

- **Steps**:
    - Start at the given node (root).
    - Mark the node as visited and enqueue it.
    - While the queue is not empty, dequeue a node, print it, and enqueue its unvisited neighbors.
- **Key Concepts**: Queue, Graph traversal.

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
bfs(graph, 'A')
```

2. Depth First Search (DFS) Implementation

DFS is another graph traversal algorithm that explores as far as possible along each branch before backtracking. It uses recursion or an explicit stack.

- **Steps**:
    - Start at the given node (root).
    - Mark it as visited.
    - Recursively visit all unvisited neighbors.
- **Key Concepts**: Recursion, Stack, Graph traversal.

```
def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
    print(node, end=" ")
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
# Example usage
dfs(graph, 'A')
```

3. Merge Sort Implementation

Merge Sort is a divide-and-conquer algorithm that divides the list into halves, sorts each half recursively, and merges them.

- **Steps**:
    - Recursively split the list into two halves until each half contains one element.
    - Merge the halves in a sorted manner.

- **Key Concepts**: Recursion, Divide and conquer, Merging.

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]
        merge_sort(left)
        merge_sort(right)
        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1
# Example usage
arr = [38, 27, 43, 3, 9, 82, 10]
merge_sort(arr)
print(arr)
```

4. Heap Sort Implementation

Heap Sort is based on the binary heap data structure. It sorts by first building a max heap (where the largest element is at the root) and then repeatedly extracting the root (maximum) element and rebuilding the heap.

- **Steps**:
    - Build a max heap.

- Swap the root (largest element) with the last element.
- Reduce the heap size and heapify the tree again.
- Repeat until the heap size is reduced to 1.

- **Key Concepts**: Heap, Binary tree, Sorting.

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def heap_sort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
# Example usage
arr = [12, 11, 13, 5, 6, 7]
heap_sort(arr)
print(arr)
```

5. Bubble Sort Implementation

Bubble Sort is a simple comparison-based algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until no swaps are needed.

- **Steps**:
  - Compare adjacent elements.
  - Swap if out of order.
  - Repeat the process until the list is sorted.

- **Key Concepts**: Swapping, Comparison.

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
# Example usage
arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print(arr)
```

6. Insertion Sort Implementation

Insertion Sort builds the final sorted array one item at a time. It picks the next element and places it in its correct position by shifting the elements that are greater.

- **Steps**:
    - Start from the second element.
    - Insert it in the correct position in the sorted part of the list.
    - Repeat until the entire list is sorted.
- **Key Concepts**: Comparison, Insertion.

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
# Example usage
arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
print(arr)
```

7. Selection Sort Implementation

Selection Sort works by repeatedly finding the minimum element from the unsorted part and swapping it with the first unsorted element.

- **Steps**:
    - Find the smallest element in the unsorted part.
    - Swap it with the first unsorted element.
    - Move the boundary between sorted and unsorted parts.
- **Key Concepts**: Selection, Comparison, Swapping.

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
# Example usage
arr = [29, 10, 14, 37, 13]
selection_sort(arr)
print(arr)
```

8. Recursion Implementation

Recursion is a method of solving problems where a function calls itself. This example calculates the factorial of a number.

- **Steps**:
    - If n == 0, return 1 (base case).
    - Otherwise, return n * factorial(n-1).
- **Key Concepts**: Base case, Recursive call.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
# Example usage
print(factorial(5))
```

9. Binary Search Implementation

Binary Search is an efficient algorithm for finding an item from a sorted list by repeatedly dividing the search interval in half. It works by comparing the target value to the middle element of the list.

- **Steps**:
    - Compare the middle element with the target.
    - If the middle element is equal to the target, return its index.
    - If the target is smaller than the middle element, search the left half.
    - If the target is greater, search the right half.
- **Key Concepts**: Divide and conquer, Logarithmic time complexity.

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1


# Example usage
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
result = binary_search(arr, 4)
print(result)
```

10. Linear Search Implementation

Linear Search sequentially checks each element in a list until it finds the target or exhausts the list.

- **Steps**:
    - Start from the first element and check if it's equal to the target.
    - If it is, return the index.
    - If it's not, move to the next element and repeat.
- **Key Concepts**: Sequential search, Time complexity is O(n).

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
```

```
            return i
    return -1
# Example usage
arr = [10, 20, 30, 40, 50]
result = linear_search(arr, 30)
print(result)
```

11. Class Implementation

This program demonstrates the creation of a class in Python. A Dog class is created with name and breed attributes, and a method bark that prints a message.

- **Steps**:
    - Define the class with an __init__ constructor.
    - Add methods to interact with object data.
- **Key Concepts**: Classes, Methods, Object-oriented programming.

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print(f"{self.name} is barking!")

# Example usage
dog = Dog("Buddy", "Golden Retriever")
dog.bark()
```

12. Inheritance Example

Inheritance allows one class (child class) to inherit the attributes and methods from another class (parent class). In this case, the Car class inherits from the Vehicle class.

- **Steps**:
    - Define a base class Vehicle.
    - Define a derived class Car that inherits from Vehicle and adds extra attributes.
- **Key Concepts**: Inheritance, Method overriding.

```
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
```

```
        self.model = model
        self.year = year
    def get_info(self):
        return f"{self.year} {self.make} {self.model}"


class Car(Vehicle):
    def __init__(self, make, model, year, doors):
        super().__init__(make, model, year)
        self.doors = doors
    def get_info(self):
        return f"{super().get_info()} with {self.doors} doors"
# Example usage
car = Car("Toyota", "Corolla", 2020, 4)
print(car.get_info())
```

---

13. Polymorphism Example

Polymorphism allows different classes to define the same method in their own way. Here, Shape is a base class, and both Circle and Rectangle override the area method.

- **Steps**:
    - Create a base class with an abstract method.
    - Derive classes and override the method to implement specific behavior.
- **Key Concepts**: Polymorphism, Method overriding, Abstraction.

```
class Shape:
    def area(self):
        pass
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius * self.radius
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

```
    def area(self):
        return self.width * self.height

# Example usage
circle = Circle(5)
rectangle = Rectangle(4, 6)
print(circle.area())
print(rectangle.area())
```

14. Bank Account Class

This program demonstrates encapsulation by creating a BankAccount class with a private attribute __balance. It provides methods to deposit, withdraw, and get the balance.

- **Steps**:
    - Define private attributes using double underscores (__).
    - Create methods to interact with those private attributes.
- **Key Concepts**: Encapsulation, Private attributes, Object-oriented programming.

```
class BankAccount:
    def __init__(self, balance=0):
        self.__balance = balance
    def deposit(self, amount):
        self.__balance += amount
    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print("Insufficient funds")

    def get_balance(self):
        return self.__balance
# Example usage
account = BankAccount()
account.deposit(1000)
account.withdraw(500)
print(account.get_balance())
```

15. Interface-like Behavior in Python

Python does not have explicit interfaces like Java, but it can mimic them using abstract base classes (ABC). This program defines an interface Reader with an abstract method read(), and two classes EBook and Magazine implement it.

- **Steps**:
  - Use the abc module to create an abstract class.
  - Define abstract methods that must be implemented by subclasses.
- **Key Concepts**: Abstract base classes, Interfaces, Polymorphism.

```
from abc import ABC, abstractmethod

class Readable(ABC):
    @abstractmethod
    def read(self):
        pass

class EBook(Readable):
    def read(self):
        return "Reading eBook"

class Magazine(Readable):
    def read(self):
        return "Reading Magazine"


# Example usage
ebook = EBook()
magazine = Magazine()
print(ebook.read())
print(magazine.read())
```

16. Multiple Inheritance

Multiple inheritance allows a class to inherit from more than one base class. Here, the Bat class inherits from both Animal and Bird.

- **Steps**:
  - Create multiple base classes.
  - Derive a class that inherits from all base classes.
- **Key Concepts**: Multiple inheritance, Method resolution order (MRO).

```
class Animal:
    def speak(self):
        print("Animal speaks")
class Bird:
    def fly(self):
```

```
        print("Bird flies")
class Sparrow(Animal, Bird):
    def chirp(self):
        print("Sparrow chirps")
# Example usage
sparrow = Sparrow()
sparrow.speak()
sparrow.fly()
sparrow.chirp()
```

17. Abstract Class Example

An abstract class is a class that cannot be instantiated and usually contains abstract methods that must be implemented in derived classes. Here, Shape is an abstract class with an abstract area method.

- **Steps**:
    - Use the abc module to define abstract methods.
    - Derived classes implement these abstract methods.
- **Key Concepts**: Abstraction, Abstract methods.

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
# Example usage
rectangle = Rectangle(5, 4)
print(rectangle.area())
```

18. Interface Example

This program demonstrates how Python can implement interface-like behavior using abstract base classes. The Readable interface has a read method, and both EBook and Magazine implement this method.

- **Steps**:
    - Define an abstract class to act as an interface.
    - Ensure subclasses implement the required methods.
- **Key Concepts**: Interfaces, Polymorphism, Abstract base classes.

```
from abc import ABC, abstractmethod
class Readable(ABC):
    @abstractmethod
    def read(self):
        pass
class EBook(Readable):
    def read(self):
        return "Reading eBook"
class Magazine(Readable):
    def read(self):
        return "Reading Magazine"
# Example usage
ebook = EBook()
magazine = Magazine()
print(ebook.read())
print(magazine.read())
```

19. Install MySQL and Connect

This program demonstrates how to install the MySQL connector and connect Python to a MySQL database.

- **Steps**:
    - Install MySQL connector using pip install mysql-connector-python.
    - Use mysql.connector.connect() to establish a connection.
- **Key Concepts**: Database connectivity, SQL queries.

```
import mysql.connector
# Establishing the connection
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="password",
    database="mydb"
```

```
)
cursor = conn.cursor()
# Creating a table
cursor.execute("CREATE TABLE users (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(100))")
# Inserting data
cursor.execute("INSERT INTO users (name) VALUES ('John')")
conn.commit()
# Fetching data
cursor.execute("SELECT * FROM users")
for row in cursor.fetchall():
    print(row)
conn.close()
```

20. Employee Database Program

This program demonstrates how to manage a database using SQLite in Python. It creates a database, creates a table, and performs CRUD operations (insert, update, delete, and select).

- **Steps**:
    - Use sqlite3 module to interact with the database.
    - Perform SQL operations like CREATE, INSERT, UPDATE, DELETE, and SELECT.
- **Key Concepts**: SQLite, Database operations, CRUD.

```
import sqlite3
# Establishing the connection
conn = sqlite3.connect('company.db')
cursor = conn.cursor()
# Creating table
cursor.execute('''
    CREATE TABLE employees (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        age INTEGER,
        department TEXT
    )
''')
# Inserting records
cursor.executemany('''
```

```
    INSERT INTO employees (name, age, department) VALUES (?, ?, ?)
''', [('John', 28, 'IT'), ('Alice', 30, 'HR'), ('Bob', 35, 'Finance')])
# Updating records
cursor.execute('''
    UPDATE employees SET age = 31 WHERE name = 'Alice'
''')
# Deleting a record
cursor.execute('''
    DELETE FROM employees WHERE name = 'Bob'
''')
# Fetching all records
cursor.execute('SELECT * FROM employees')
print(cursor.fetchall())

conn.commit()
conn.close()
```