Selection Sort: Selection sort is another simple sorting algorithm that divides the input list into two parts: a sorted sublist and an unsorted sublist.

Initially, the sorted sublist is empty, and the unsorted sublist contains all elements.

The algorithm finds the smallest (or largest, depending on sorting order) element in the unsorted sublist and swaps it with the leftmost unsorted element.

After each iteration, the sorted sublist grows, and the unsorted sublist shrinks until no elements remain.

Selection sort has a time complexity of O(n^2), similar to bubble sort, but it typically performs fewer swaps.

Both of these sorting algorithms are considered elementary because they are straightforward and easy to understand, although they may not be the most efficient for large datasets. They are often used for educational purposes or for sorting small datasets where simplicity is valued over performance.

Example:

```
def selection_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # Find the minimum element in the remaining
unsorted array
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
```

```
        # Swap the found minimum element with the first
element
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

# Example usage
array = [64, 34, 25, 12, 22, 11, 90]
selection_sort(array)
print("Sorted array using Selection Sort:", array)
```

Searching algorithms

Searching algorithms are used to find the presence or absence of a target value within a collection of data. Here are two commonly used searching algorithms:

Linear Search: Linear search is a simple searching algorithm that sequentially checks each element in a collection until the target value is found or all elements have been checked.

It works well for small datasets or unsorted collections.

The time complexity of linear search is O(n) in the worst-case scenario, where n is the number of elements in the collection.

Example:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return the index of the target if
found
    return -1  # Return -1 if the target is not found
```

```
# Example usage
array = [4, 2, 7, 1, 9, 5]
target = 7
result = linear_search(array, target)
if result != -1:
    print(f"Element {target} is present at index {result}")
else:
    print(f"Element {target} is not present in the array")
```

Binary Search: Binary search is a more efficient searching algorithm applicable only to sorted collections.

It works by repeatedly dividing the sorted collection in half and comparing the target value with the middle element.

If the target value matches the middle element, the search is successful. If the target value is less than the middle element, the search continues on the lower half of the collection; otherwise, it continues on the upper half.

Binary search has a time complexity of O(log n), making it significantly faster than linear search for large datasets.

Example:

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid  # Return the index of the target if
found
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1  # Return -1 if the target is not found

# Example usage
sorted_array = [1, 2, 3, 4, 5, 6, 7, 8, 9]
target = 7
result = binary_search(sorted_array, target)
if result != -1:
    print(f"Element {target} is present at index {result}")
else:
    print(f"Element {target} is not present in the array")
```

Mathematical Algorithms:

Factorial: Calculates the factorial of a non-negative integer n, denoted by n!, which is the product of all positive integers less than or equal to n.

Example:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

# Example usage:
n = 5
print("Factorial of", n, "is", factorial(n))
```

Fibonacci Sequence: Generates the Fibonacci sequence up to the nth term, where each term is the sum of the two preceding terms.

Example:

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

# Example usage:
n = 7
print("Fibonacci sequence up to", n, "terms:")
for i in range(n):
    print(fibonacci(i), end=" ")
```

String Algorithms:

Reverse a String: Reverses a given string s by slicing it with a step of -1.

Example:

```
def reverse_string(s):
    return s[::-1]

# Example usage:
s = "hello"
print("Original:", s)
print("Reversed:", reverse_string(s))
```

Check Palindrome: Determines whether a given string s is a palindrome, which reads the same forwards and backwards.

Example:

```
def is_palindrome(s):
    return s == s[::-1]

# Example usage:
s = "radar"
print("Is", s, "a palindrome?", is_palindrome(s))
```

Recursion:

Sum of Digits: Computes the sum of digits of a non-negative integer n using recursion.

Example:

```
def sum_of_digits(n):
    if n == 0:
        return 0
    else:
        return n % 10 + sum_of_digits(n // 10)

# Example usage:
n = 12345
print("Sum of digits in", n, "is", sum_of_digits(n))
```

Greatest Common Divisor (GCD): Finds the greatest common divisor of two non-negative integers a and b using the Euclidean algorithm and recursion.

Example:

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)

# Example usage:
a, b = 36, 24
print("GCD of", a, "and", b, "is", gcd(a, b))
```

Introduction to Data Structures

Data structures are a fundamental concept in computer science that enable efficient organization, storage, and manipulation of data. They provide a way to represent and manage collections of data in a structured manner, allowing for optimized access, insertion, deletion, and searching operations.

Here's an overview of some common data structures:

- **Stack**
- **Queue**
- **Linked List**

Stack

A stack is a fundamental data structure that follows the Last In, First Out (LIFO) principle. It is named "stack" because it resembles a stack of plates, where you can only add or remove plates from the top.

Here are some key features and operations of a stack:

Features:

- A stack is a collection of elements with two main operations: push and pop.
- Elements are added or removed from the top of the stack.
- Stacks are often implemented using arrays or linked lists.

Operations:

- **Push:** Adds an element to the top of the stack.
- **Pop:** Removes and returns the top element from the stack.
- **Peek (or Top):** Returns the top element of the stack without removing it.
- **isEmpty:** Checks if the stack is empty.
- **Size:** Returns the number of elements in the stack.

Applications:

- Stacks are commonly used in programming languages for function call management and maintaining the call stack.

- They are used in expression evaluation algorithms (e.g., infix to postfix conversion, postfix evaluation).

- Undo functionality in text editors and command-line interfaces often uses stacks to keep track of previous states or commands.

- Backtracking algorithms and depth-first search (DFS) traversal utilize stacks to keep track of visited nodes or search paths.

Implementation:

- Stacks can be implemented using arrays or linked lists.

- Array-based implementations are simpler and provide constant-time access to the top element, but may require resizing if the stack grows beyond its initial capacity.

- Linked list-based implementations allow for dynamic memory allocation and efficient resizing, but may have slightly higher overhead due to pointer manipulation.

Here's a simple implementation of a stack in Python using a list:

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
```

```
        if not self.isEmpty():
            return self.items.pop()
        else:
            raise IndexError("Stack is empty")

    def peek(self):
        if not self.isEmpty():
            return self.items[-1]
        else:
            raise IndexError("Stack is empty")

    def size(self):
        return len(self.items)
```

In this implementation, push() adds an element to the top of the stack, pop() removes and returns the top element, peek() returns the top element without removing it, isEmpty() checks if the stack is empty, and size() returns the number of elements in the stack.

Queue

A queue is another fundamental data structure that follows the First In, First Out (FIFO) principle. It is similar to a line of people waiting to be served, where the person who has been waiting the longest is served first.

Here are some key features and operations of a queue:

Features:

- A queue is a collection of elements with two main operations: enqueue and dequeue.

- Elements are added to the rear (enqueue) and removed from the front (dequeue) of the queue.
- Queues are often implemented using arrays or linked lists.

Operations:

- **Enqueue:** Adds an element to the rear of the queue.
- **Dequeue:** Removes and returns the element from the front of the queue.
- **Front:** Returns the front element of the queue without removing it.
- **isEmpty:** Checks if the queue is empty.
- **Size:** Returns the number of elements in the queue.

Applications:

- Queues are commonly used in scheduling algorithms, such as job scheduling in operating systems.
- They are used in breadth-first search (BFS) traversal algorithms for graph traversal.
- Print queues in computer systems use queues to manage the order of print jobs.
- Buffer management in networking and I/O systems often employs queues to manage data flow.

Implementation:

- Queues can be implemented using arrays or linked lists.
- Array-based implementations are simple and provide constant-time access to both the front and rear of the queue, but may require resizing if the queue grows beyond its initial capacity.
- Linked list-based implementations allow for dynamic memory allocation and efficient resizing, but may have slightly higher overhead due to pointer manipulation.

Here's a simple implementation of a queue in Python using a list:

```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return len(self.items) == 0

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.isEmpty():
            return self.items.pop(0)
        else:
            raise IndexError("Queue is empty")

    def front(self):
        if not self.isEmpty():
            return self.items[0]
        else:
            raise IndexError("Queue is empty")
```

```
    def size(self):
        return len(self.items)
```

In this implementation, enqueue() adds an element to the rear of the queue, dequeue() removes and returns the element from the front of the queue, front() returns the front element without removing it, isEmpty() checks if the queue is empty, and size() returns the number of elements in the queue.

Linked List

A linked list is a linear data structure where elements are stored in memory as separate objects called nodes. Each node contains both data and a reference (or pointer) to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory allocation, allowing for dynamic memory allocation and efficient insertion and deletion operations.

Here are some key features and operations of a linked list:

Features:

- A linked list consists of nodes, where each node contains data and a reference to the next node in the sequence.
- The last node in the list typically points to None, indicating the end of the list.
- Linked lists can be singly linked (each node points to the next node) or doubly linked (each node points to both the next and previous nodes).

Operations:

Insertion:

- **At the beginning:** Insert a new node at the beginning of the list.
- **At the end:** Insert a new node at the end of the list.
- **In the middle:** Insert a new node at a specific position in the list.

Deletion:

- **At the beginning:** Remove the first node from the list.
- **At the end:** Remove the last node from the list.
- **In the middle:** Remove a node from a specific position in the list.
- **Traversal:** Traverse the list to access or modify each node's data.
- **Search:** Search for a specific value in the list.

Types of Linked Lists:

- **Singly Linked List:** Each node has a reference to the next node in the sequence.
- **Doubly Linked List:** Each node has references to both the next and previous nodes in the sequence.
- **Circular Linked List**: The last node in the list points back to the first node, forming a circular structure.

Applications:

- Linked lists are often used when the size of the data structure is not known in advance or when frequent insertions and deletions are required.

- They are used in various applications, such as implementing stacks, queues, hash tables, and adjacency lists for graphs.

Implementation:

- Linked lists can be implemented using classes and pointers (references) in languages like Python, Java, C++, and others.
- Each node in the linked list contains two fields: one for data and another for the reference to the next node.

Here's a simple implementation of a singly linked list in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
```

```
            last_node = self.head
            while last_node.next:
                last_node = last_node.next
            last_node.next = new_node

    def display(self):
            current = self.head
            while current:
                print(current.data, end=" ")
                current = current.next
            print()

# Example usage:
ll = LinkedList()
ll.insert_at_beginning(1)
ll.insert_at_end(2)
ll.insert_at_end(3)
ll.display()  # Output: 1 2 3
```

In this implementation, each node of the linked list is represented using the Node class, and the linked list itself is represented using the **LinkedList** class. You can insert nodes at the beginning or end of the list and display the contents of the list.

More Data Structures

Hash table

A hash table (hash map) is a data structure that stores key-value pairs and provides efficient insertion, deletion, and lookup operations.

It uses a hash function to compute an index (hash code) for each key, allowing for constant-time access to values associated with keys.

Hash tables are widely used in applications such as implementing associative arrays, symbol tables, and caching.

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value  # Update existing key
                return
        self.table[index].append([key, value])  # Add new
key-value pair

    def get(self, key):
        index = self._hash(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]  # Return value associated
with key
```

```
        return None  # Key not found

    def remove(self, key):
        index = self._hash(key)
        for i, pair in enumerate(self.table[index]):
            if pair[0] == key:
                del self.table[index][i]  # Remove key-
value pair
                return
        raise KeyError(f'Key "{key}" not found')

# Example usage:
hash_table = HashTable(10)
hash_table.insert('apple', 5)
hash_table.insert('banana', 10)
hash_table.insert('orange', 15)

print("Value for key 'apple':", hash_table.get('apple'))  #
Output: 5
print("Value for key 'banana':",
hash_table.get('banana'))  # Output: 10
print("Value for key 'orange':",
hash_table.get('orange'))  # Output: 15

hash_table.insert('apple', 7)  # Update value for existing
key
print("Updated value for key 'apple':",
hash_table.get('apple'))  # Output: 7

hash_table.remove('banana')
print("Value for key 'banana' after removal:",
hash_table.get('banana'))  # Output: None
```