AI-ASSISTED CODING

ASSIGNMENT-12.1

Algorithms with AI Assistance – Sorting, Searching, and  
Optimizing Algorithms

**NAME: GATTU VARSHITH**

**HALL-TICKET: 2403A510E3**

**BATCH NO: 05**

**TASK-1**:  
Use AI to generate a Python program that implements the  
Merge Sort algorithm.  
• Instructions:  
o Prompt AI to create a function merge\_sort(arr) that sorts a  
list in ascending order.  
o Ask AI to include time complexity and space complexity  
in the function docstring.  
o Verify the generated code with test cases.

**Prompt:**

Make a Python program with a function merge\_sort(arr) that sorts a list. Add a docstring with time and space complexity. Test it with some example lists.

**CODE GENERATED:**

def merge\_sort(arr):

    """

    Merge Sort Algorithm

    This function sorts a list in ascending order using Merge Sort.

    Steps:

    1. Divide: Split the list into two halves.

    2. Conquer: Recursively sort both halves.

    3. Combine: Merge the sorted halves.

    Time Complexity: O(n log n)  # Works the same for best, average, and worst cases

    Space Complexity: O(n)       # Extra space needed for temporary arrays

    """

    if len(arr) <= 1:

        return arr

    mid = len(arr) // 2

    left\_half = merge\_sort(arr[:mid])

    right\_half = merge\_sort(arr[mid:])

    return merge(left\_half, right\_half)

def merge(left, right):

    """Helper function to merge two sorted lists into one sorted list"""

    merged = []

    i = j = 0

    while i < len(left) and j < len(right):

        if left[i] < right[j]:

            merged.append(left[i])

            i += 1

        else:

            merged.append(right[j])

            j += 1

    # Add any remaining elements

    merged.extend(left[i:])

    merged.extend(right[j:])

    return merged

# -------- Test Cases --------

if \_\_name\_\_ == "\_\_main\_\_":

    print("Test Case 1:", merge\_sort([]))                       # []

    print("Test Case 2:", merge\_sort([5]))                      # [5]

    print("Test Case 3:", merge\_sort([1, 2, 3, 4]))             # [1, 2, 3, 4]

    print("Test Case 4:", merge\_sort([4, 3, 2, 1]))             # [1, 2, 3, 4]

    print("Test Case 5:", merge\_sort([38, 27, 43, 3, 9, 82, 10]))

    # [3, 9, 10, 27, 38, 43, 82]

**OUTPUT:**

**A black background with white numbers

AI-generated content may be incorrect.**

**OBSERVATION :**

Merge Sort is a divide-and-conquer algorithm. It splits the list into smaller parts, sorts them, and then merges them back. The time complexity is **O(n log n)** in all cases, which makes it efficient. The space complexity is **O(n)** because extra memory is used while merging.

TASK -2:

Use AI to create a binary search function that finds a target  
element in a sorted list.  
• Instructions:  
o Prompt AI to create a function binary\_search(arr, target)  
returning the index of the target or -1 if not found.  
o Include docstrings explaining best, average, and worst-  
case complexities.  
o Test with various inputs.

PROMPT:

Create a Python function binary\_search(arr, target) that finds a target element in a sorted list and returns its index, or -1 if not found. Include a docstring explaining best, average, and worst-case time complexities and space complexity. Test the function with different inputs, including empty, single-element, and larger sorted lists.

CODE GENERATED:

def binary\_search(arr, target):

    """

    Binary Search Algorithm

    This function searches for a target element in a sorted list using

    the Binary Search algorithm. It returns the index of the target

    element if found, otherwise it returns -1.

    Parameters:

    -----------

    arr : list

        A sorted list of elements in which to search.

    target : int or float

        The element to search for in the list.

    Returns:

    --------

    int

        Index of the target element if found, otherwise -1.

    Time Complexity:

    ----------------

    Best Case: O(1)       -> Target is at the middle element.

    Average Case: O(log n)

    Worst Case: O(log n)

    Space Complexity: O(1) -> Constant space used.

    Example:

    --------

    >>> binary\_search([1, 3, 5, 7, 9], 7)

    3

    >>> binary\_search([2, 4, 6, 8], 5)

    -1

    """

    left, right = 0, len(arr) - 1

    while left <= right:

        mid = (left + right) // 2

        if arr[mid] == target:

            return mid

        elif arr[mid] < target:

            left = mid + 1

        else:

            right = mid - 1

    return -1

# -------- Test Cases --------

if \_\_name\_\_ == "\_\_main\_\_":

    test\_cases = [

        ([1, 3, 5, 7, 9], 7),

        ([2, 4, 6, 8, 10, 12], 12),

        ([10, 20, 30, 40], 25),

        ([5], 5),

        ([], 3)

    ]

    for lst, target in test\_cases:

        index = binary\_search(lst, target)

        print(f"Element {target} index is {index}")

OUTPUT:  
A screen shot of a computer screen

AI-generated content may be incorrect.

OBSERVATION:

Binary Search searches for an element in a sorted list. It repeatedly divides the list to narrow down the search. It returns the index of the element if found, otherwise -1.

TASK – 3:

• Scenario: A retail store’s inventory system contains thousands of  
products, each with attributes like product ID, name, price, and  
stock quantity. Store staff need to:  
1. Quickly search for a product by ID or name.  
2. Sort products by price or quantity for stock analysis.  
• Task:  
o Use AI to suggest the most efficient search and sort

algorithms for this use case.  
o Implement the recommended algorithms in Python.  
o Justify the choice based on dataset size, update frequency,  
and performance requirements.

PROMPT:

Create a Python program to manage store products with ID, name, price, and quantity.  
Add functions to search by ID or name and sort by price or quantity. Include docstrings and test the functions with sample data.

CODE GENERATED:

# -------- Dataset --------

products = [

    {"id": 101, "name": "Laptop", "price": 800, "quantity": 15},

    {"id": 102, "name": "Mouse", "price": 20, "quantity": 150},

    {"id": 103, "name": "Keyboard", "price": 50, "quantity": 80},

    {"id": 104, "name": "Monitor", "price": 200, "quantity": 40},

    {"id": 105, "name": "Printer", "price": 120, "quantity": 30},

    {"id": 106, "name": "USB Drive", "price": 10, "quantity": 500},

    {"id": 107, "name": "Webcam", "price": 70, "quantity": 60},

    {"id": 108, "name": "Headphones", "price": 60, "quantity": 120},

    {"id": 109, "name": "Speakers", "price": 90, "quantity": 50},

    {"id": 110, "name": "External HDD", "price": 100, "quantity": 35},

]

# -------- Efficient Search --------

# 1. Dictionary for search by ID

product\_dict = {product["id"]: product for product in products}

def search\_by\_id(product\_id):

    """

    Search for a product by its ID using a dictionary lookup.

    Parameters:

    -----------

    product\_id : int

        The ID of the product to search for.

    Returns:

    --------

    dict or None

        Returns the product dictionary if found, otherwise None.

    Time Complexity: O(1) average case

    Space Complexity: O(n) for storing the dictionary

    """

    return product\_dict.get(product\_id, None)

# 2. Binary Search for search by name (requires sorted list by name)

products\_sorted\_by\_name = sorted(products, key=lambda x: x["name"])

def binary\_search\_name(arr, target\_name):

    """

    Search for a product by its name using binary search.

    Parameters:

    -----------

    arr : list

        A list of products sorted by 'name'.

    target\_name : str

        The name of the product to search for.

    Returns:

    --------

    dict or None

        Returns the product dictionary if found, otherwise None.

    Time Complexity:

        Best Case: O(1)

        Average Case: O(log n)

        Worst Case: O(log n)

    Space Complexity: O(1)

    """

    left, right = 0, len(arr) - 1

    while left <= right:

        mid = (left + right) // 2

        if arr[mid]["name"] == target\_name:

            return arr[mid]

        elif arr[mid]["name"] < target\_name:

            left = mid + 1

        else:

            right = mid - 1

    return None

# -------- Sorting --------

def sort\_products\_by\_price(products):

    """

    Sort the products by price in ascending order.

    Parameters:

    -----------

    products : list

        List of product dictionaries.

    Returns:

    --------

    list

        Sorted list of products by price.

    Time Complexity: O(n log n)

    Space Complexity: O(n)

    """

    return sorted(products, key=lambda x: x["price"])

def sort\_products\_by\_quantity(products):

    """

    Sort the products by quantity in ascending order.

    Parameters:

    -----------

    products : list

        List of product dictionaries.

    Returns:

    --------

    list

        Sorted list of products by quantity.

    Time Complexity: O(n log n)

    Space Complexity: O(n)

    """

    return sorted(products, key=lambda x: x["quantity"])

# -------- Testing --------

if \_\_name\_\_ == "\_\_main\_\_":

    # Search Tests

    print("Search by ID 105:", search\_by\_id(105))

    print("Search by Name 'Monitor':", binary\_search\_name(products\_sorted\_by\_name, "Monitor"))

    print("Search by ID 999:", search\_by\_id(999))  # Not found

    print("Search by Name 'Tablet':", binary\_search\_name(products\_sorted\_by\_name, "Tablet"))  # Not found

    # Sort Tests

    print("\nProducts sorted by price:")

    for p in sort\_products\_by\_price(products):

        print(f"{p['name']} - Price: {p['price']}, Quantity: {p['quantity']}")

    print("\nProducts sorted by quantity:")

    for p in sort\_products\_by\_quantity(products):

        print(f"{p['name']} - Quantity: {p['quantity']}, Price: {p['price']}")

OUTPUT:

A screenshot of a computer

AI-generated content may be incorrect.

OBSERVATION:

Binary Search and dictionary lookups help find products quickly in a sorted or indexed list. Sorting products by price or quantity makes stock analysis easier and faster. These methods are efficient and work well for a moderate number of products.