MODULE 2: ALGORITHMIC THINKING

Session 2: Fundamentals of coding and algorithms 2

Searching and Sorting Algorithms

Searching and Sorting Algorithms

Searching and sorting are fundamental operations in computer science and are integral to many algorithms and data processing tasks. Understanding these algorithms is crucial for efficient data retrieval and organization.

Types of Searching and Sorting Algorithms

1. Searching Algorithms:

Searching algorithms are used to find the position of an element in a data structure (like an array or list).

Here are some common searching algorithms:

1.1 Linear Search:

A straightforward method that checks each element in a list sequentially until the desired element is found or the list ends.

- Time Complexity: O(n), where n is the number of elements in the list.
- Use Case: Suitable for small or unsorted lists

Example:

```
def linear_search(arr, target):
        for i in range(len(arr)):
            if arr[i] == target:
                return i
        return -1

    # Example usage
    arr = [10, 23, 45, 70, 11, 15]
    target = 70
    print(linear_search(arr, target))  # Output: 3
```

Explanation:

Linear search sequentially checks each element in the list to find the target value. In this example, the algorithm finds the target 70 at index 3. The time complexity of linear search is O(n), meaning the time it takes to complete the search grows linearly with the size of the list.

1.2 Binary Search

An efficient algorithm that repeatedly divides a sorted list in half, reducing the search area until the target element is found or the list is exhausted.

- **Time Complexity:** O(log n), where n is the number of elements in the list.
- **Use Case:** Suitable for large, sorted lists

Example:

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

# Test the function
arr = [2, 3, 4, 7, 9]
print(binary_search(arr, 7))  #
Output: 3
```

Explanation:

The algorithm checks the middle element, and if it matches the target, it returns the index. If the target is smaller, it continues searching in the left half; if larger, in the right half. In this example, the function successfully finds the target “7” at index “3”. The time complexity of binary search is O(log n), making it much faster than linear search for large lists.

2. Sorting Algorithms

Sorting algorithms arrange the elements of a list or array in a specific order, typically ascending or descending.

Here are some commonly used sorting algorithms:

2.1 Bubble Sort

A simple comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

- **Time Complexity:** O(n^2) in the worst and average cases.
- **Use Case:** Educational purposes or small datasets.

Example:

```
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("Sorted array is:", arr)
# Output: [11, 12, 22, 25,
34, 64, 90]
```

Explanation:

The function “`bubble_sort`” sorts the list “`[64, 34, 25, 12, 22, 11, 90]`” in ascending order. The time complexity of Bubble Sort is $O(n^2)$, making it inefficient for large lists compared to more advanced algorithms like quicksort or mergesort.

2.2 Insertion Sort

Builds the final sorted array one item at a time by repeatedly picking the next item and inserting it into its correct position among the previously sorted items.

- **Time Complexity:** O(n^2) in the worst case.
- **Use Case:** Efficient for small or nearly sorted datasets

Example:

```
def insertion_sort(arr):
    # Traverse through 1 to len(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("Sorted array is:", arr)  #
Output: [5, 6, 11, 12, 13]
```

Explanation:

The function `insertion_sort` sorts the list `[12, 11, 13, 5, 6]` in ascending order. The algorithm is efficient for small datasets or nearly sorted data, with a time complexity of $O(n^2)$ in the worst case.

2.3 Merge Sort

A divide-and-conquer algorithm that divides the list into smaller sublists, sorts them, and then merges the sorted sublists to produce the final sorted list.

- **Time Complexity:** O(n log n) in all cases.
- **Use Case:** Suitable for large datasets.

Example:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

# Example usage
arr = [38, 27, 43, 3, 9, 82, 10]
sorted_arr = merge_sort(arr)
print("Sorted array is:",
sorted_arr)  # Output: [3, 9,
10, 27, 38, 43, 82]
```

2.4 Quick Sort

A **divide-and-conquer** algorithm that works by selecting a "pivot" element and partitioning the array around the pivot. The elements smaller than the pivot go to one side, and the elements larger than the pivot go to the other side. The process is then repeated recursively for each partition.

- **Time Complexity:** O(n log n) in all cases.
- **Use Case:** Sorting large datasets, efficient average-case sorting

```python
def partition(arr, low, high):
    # Choose the pivot as the last element
    pivot = arr[high]
    i = low - 1  # Index of the smaller element

    # Rearrange elements so that those smaller than the pivot are on the left
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]  # Swap

    # Place the pivot in its correct position
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1  # Return the partition index
```

```python
def quick_sort_in_place(arr, low, high):
    if low < high:
        # Partition the array and get the pivot index
        pi = partition(arr, low, high)

        # Recursively sort elements before and after the partition
        quick_sort_in_place(arr, low, pi - 1)
        quick_sort_in_place(arr, pi + 1, high)

# Example usage
arr = [10, 80, 30, 90, 40, 50, 70]
quick_sort_in_place(arr, 0, len(arr) - 1)
print(f"Sorted array: {arr}")
```

Activity

You are given an unsorted array of integers:

arr = [33, 14, 27, 35, 10, 19, 42, 44]

You are required to implement **five different sorting algorithms** to sort this array in ascending order.

Questions to Answer:

- Write Python functions for each of the above sorting algorithms?
- Compare and discuss the time complexity and space complexity of each sorting algorithm?
- Which algorithm performs the best when the array is already sorted?

PAUSE 5 mins

Activity Feedback

Question 2: Merge Sort and **Quick Sort** both have an average-case time complexity of **O(n log n)**, making them the most efficient for typical inputs.

Question 3: Insertion Sort performs best when the array is already sorted with a time complexity of **O(n)**. It only compares adjacent elements and does not require any swaps, so it's highly efficient for nearly sorted arrays.

- This marks the end of Module 2.
- In the next Module we will learn about Project-based learning.

THANK YOU