Hyperiondev

TASK

Sorting and Searching

Model-Answer Approach

Visit our website

Auto-graded task 1

album_management.py

This approach defines a class `Album` to represent a music album, encapsulating attributes like `album_name`, `album_artist`, and `number_of_songs`. It provides methods for initialising and converting an album instance to a string format. The program also includes several utility functions: `display_albums` for printing album details, `sort_number_of_songs` for sorting albums by song count, `swap_positions` for exchanging the positions of two albums in a list, and `find_index` for locating the index of an album by its name. An example usage demonstrates these functionalities on two lists of albums (`albums1` and `albums2`), showcasing sorting, swapping, and merging operations, along with adding new albums and sorting them alphabetically.

While the code is functional, there are some potential pitfalls that can be considered to improve the efficiency of the code. First, the `swap_positions` function assumes that the provided positions are within the valid range of the list, which can lead to errors if they are out of bounds. The code also uses integer indices without validation, which can cause unexpected behaviour if the list structure changes or if user input is incorrect. Additionally, the use of string literals for album names in functions like `find_index` can lead to issues if there are typos or case mismatches. Lastly, the `sort` method is used with a lambda function to sort by album name, which could be made more robust by handling case sensitivity or providing more complex sorting criteria. These issues could be mitigated by adding error handling, input validation, and more flexible comparison logic.

Auto-graded task 2

merge_sort.py

The modified merge sort algorithm orders a list of strings by their lengths from longest to shortest using a divide-and-conquer strategy. The algorithm begins by determining the length of the input list and creates a temporary storage array for merging. It then iteratively doubles the size of subsections to be merged until the entire list is processed. During each iteration, the list is divided into smaller sections, and the `merge_by_length` function compares the lengths of strings from two sections, placing the longer string into the temporary storage. After merging,

the sorted elements are copied back to the original list. This process continues until the entire list is sorted.

While the modified merge sort algorithm effectively sorts strings by length, it has several shortfalls. Firstly, the algorithm's complexity can lead to inefficiencies with very large lists due to the multiple iterations and recursive merging steps, consuming significant memory for temporary storage. Additionally, the merge function's implementation assumes non-empty lists and doesn't handle edge cases like empty lists or lists with identical string lengths gracefully. Lastly, the algorithm focuses solely on string length, ignoring lexicographic order among strings of the same length, which may be a requirement in some applications.

Auto-graded task 3

sort_search.py

The approach provides implementations for three fundamental algorithms: linear search, insertion sort, and binary search. The `linear_search` function iterates through a list of numbers to find a target number, returning the index if found or `-1` otherwise. This approach is suitable for unsorted lists. The `insertion_sort` function sorts a list of numbers in ascending order by repeatedly inserting elements into their correct positions, modifying the list in place. After sorting, the `binary_search` function is employed, which efficiently finds the target number in a sorted list by repeatedly dividing the search interval in half, leveraging the sorted property of the data for quick lookup. The code includes example usage, first demonstrating a linear search on an unsorted list, followed by sorting the list, and then performing a binary search to locate the target number.

A possible pitfall is that the `linear_search` function is inefficient for large lists due to its `O(n)` time complexity, especially when used on unsorted data, which is typical for this algorithm. The `insertion_sort` method, while stable and simple, has a time complexity of `O(n^2)`, making it impractical for large datasets compared to more efficient sorting algorithms like quicksort or mergesort. Furthermore, the `binary_search` function assumes the list is sorted; if used incorrectly on unsorted data, it could yield incorrect results or cause logical errors. Another limitation is that these functions handle only numerical data; for more complex data types, additional handling or custom comparison functions would be required. To

mitigate these issues, choosing the right algorithm based on the data size and characteristics, and ensuring data is appropriately prepared for the operations, is crucial.