**Report on Algorithms & Constructs**

**Sorting Algorithm: Merge Sort**

Merge sort has been employed in the system to sort the list of people. Merge sort is chosen due to its stable performance across various scenarios. Its time complexity is consistently O (n log n) making it efficient for large datasets.

The merge sort algorithm divides the array into smaller sub-arrays, sorts them individually, and then merges them back together. This divide-and-conquer approach ensures that the sorting process is efficient and reliable. Additionally, merge sort is suitable for sorting objects with complex data structures, like the Person objects in this code, as it only requires comparisons between elements.

**Advantages**

* **Consistent Performance:** Merge sort performs consistently well even with large datasets, thanks to its time complexity of O (n log n).
* **Stability:** Merge sort is a stable sorting algorithm, meaning it preserves the relative order of equal elements. This property is crucial when sorting complex objects based on multiple criteria.
* **Suitability for Linked Lists:** Merge sort performs efficiently on linked lists due to its ability to handle sequential access patterns.

**Comparison**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Space Complexity** | **Stability** | **Suitability** | **Comparison** |
| **Merge Sort** | **O (n log n)**  In all cases | **O (n)**  For auxiliary array. | **Yes** | Large datasets, linked lists, and objects with complex structures. | Merge sort consistently performs well with its time complexity, making it suitable for general-purpose sorting. |
| **Quick Sort** | **O (n log n)** (average case)  **O (n^2)**  (worst case) | **O (log n)**  Due to recursive calls | **No** | Large datasets, especially when average case performance matters more than worst-case scenarios. | Quick sort can outperform merge sort in practice due to lower constant factors, but its worst-case behavior makes it less desirable for critical applications. |
| **Insertion Sort** | **O (n^2) (average & worst case)** | **O (1)** | **Yes** | Small datasets or nearly sorted lists. | Insertion sort is simple and efficient for small datasets but becomes inefficient for larger ones due to its quadratic time complexity. |

**Conclusion**

Merge sort offers consistent performance and stability, making it suitable for general use. Quick sort can outperform merge sort in practice but has a higher risk of worst-case behavior. Insertion sort is efficient for small datasets but impractical for larger ones due to its quadratic time complexity.

**Searching Algorithm: Linear Search**

Linear search is utilized in the program to find a person by name within the list of people. Although linear search is not the most efficient search algorithm for large datasets, it is suitable for this context because the list of people is expected to be relatively small.

**Advantages**

* **Simplicity:** Linear search is straightforward to implement and understand, making it suitable for small-scale applications.
* **Flexibility:** Linear search can be applied to both sorted and unsorted lists without any preprocessing, making it versatile.
* **Minimal Memory Usage:** Linear search requires minimal additional memory beyond the list being searched, making it memory efficient.

**Comparison**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Space Complexity** | **Suitability** | **Comparison** |
| **Linear Search** | **O (n)**  In worst case | **O (1)** | Small datasets or when the element to be found is likely to be near the beginning of the list. | Linear search is straightforward but inefficient for large datasets since it scans each element sequentially. |
| **Binary Search** | **O (log n)**  (For sorted Arrays) | **O (1)** | Sorted arrays or lists. | Binary search is highly efficient for large, sorted datasets, but it requires the array to be sorted beforehand, which may incur additional overhead. |
| **Hashing**  **(Hash Table)** | **O (1)** | **O (n)** | Retrieving elements quickly with known keys. | Hashing provides constant-time search in average cases but requires additional space for hashing functions and collision resolution. |

**Conclusion**

Linear search is simple but inefficient for large datasets. Binary search is highly efficient but requires sorted data. Hashing provides constant-time search but requires additional space. The choice depends on factors such as dataset size, structure, and search requirements.