Written Exercise for Lab05 Merge Sort

Name: Rong Huang
Date:02/27/2023

1. Explain what you think the worst-case, Big-O complexity and the best-case, Big-O complexity of merge sort is. Why do you think that?

Worst-Case Complexity:
The worst-case time complexity of merge sort is O(n log n). This is because, regardless of the input array's initial order, merge sort divides the array into halves recursively until each subset contains only one element. Then, it merges these subsets back together in sorted order. Each division reduces the size of the problem by half (which contributes the log n part), and for each level of division, all n elements need to be merged back (contributing the n part). Therefore, the process leads to a total time complexity of O(n log n).

Best-Case Complexity:
Interestingly, the best-case time complexity for merge sort also remains O(n log n). Unlike some other sorting algorithms that can benefit from the initial order of the input (such as insertion sort, which can achieve O(n) complexity when the input array is already sorted), merge sort performs the same number of operations regardless of the input's initial order. It still divides the entire array and merges them back, which inherently involves going through all the elements multiple times in a structured manner, thus retaining the O(n log n) complexity.

Why do you think that?
This consistent behavior in both worst-case and best-case scenarios underscores the inherent efficiency and robustness of merge sort for a wide range of input conditions. Unlike algorithms that can exploit properties of the input (e.g., already sorted arrays), merge sort's divide-and-conquer strategy ensures a stable and predictable performance. This makes merge sort particularly suitable for applications where the input's initial order is unpredictable or varies widely. The O(n log n) complexity reflects the algorithm's systematic approach to sorting, balancing the depth of division (log n) with the breadth of merging operations (n) across all levels of division.

2. Explain what you think the worst-case, Big-O complexity and the best-case, Big-O complexity is for this iterative merge sort. Why do you think that?

Worst-Case Complexity:

Just like its recursive counterpart, the worst-case complexity of iterative merge sort is O(n log n). This is because the algorithm still divides the array into smaller segments and then merges them in a sorted manner. The division is done iteratively, with the size of segments doubling each time (from 1 to 2, 2 to 4, and so on), which directly mirrors the logarithmic division in recursive merge sort. Merging these segments involves comparing elements and arranging them, which takes linear time relative to the total number of elements across all segments at each level of merging. Thus, the overall process requires O(n log n) time due to the log n levels of merging and the linear work done at each level.

Best-Case Complexity:

For iterative merge sort, the best-case complexity also remains O(n log n) for similar reasons. Regardless of the initial order of the array, the algorithm must still perform the same iterative divisions and subsequent merging operations. Each segment, starting from individual elements, is merged with its adjacent segment in a manner that requires all elements to be considered and placed accordingly. Therefore, the structured and methodical process of division and merging ensures that the time complexity does not improve beyond O(n log n), even in the best-case scenario.

Why do you think that?

The reason for the consistent O(n log n) complexity in both worst-case and best-case scenarios for iterative merge sort lies in its fundamental operation principle. The algorithm's efficiency and stability stem from its divide-and-conquer approach, applied iteratively in this variant. By systematically doubling the size of segments to be merged, the algorithm ensures that each element is part of a sorted merge operation multiple times across different levels of the process. This guarantees that, irrespective of the array's initial state, every part of the array is addressed in a uniformly efficient manner, leading to the O(n log n) complexity. This complexity reflects the balanced nature of divide-and-conquer algorithms, where the division (logarithmic in nature) and the linear merging operations combine to offer optimized performance for sorting tasks.