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

Answer:

Worst-case Big-O complexity ($O(n^2)$):

Think of a long list of numbers that are in reverse order, like [5, 4, 3, 2, 1]. Bubble sort works by repeatedly comparing adjacent pairs of numbers and swapping them if they're in the wrong order. In the worst case, it needs to do this for every pair in the list, resulting in a lot of comparisons and swaps. With each pass through the list, the largest unsorted element "bubbles" to its correct position at the end of the list. So, if you have n elements, it might need to make n passes through the list, and for each pass, it needs to compare every pair of numbers.

Best-case Big-O complexity ($O(n)$):

Now, imagine the list is already sorted, like [1, 2, 3, 4, 5]. In this case, bubble sort still needs to go through the list once to confirm that it's sorted. But since it doesn't need to swap any elements, it can finish after just one pass. So, in the best-case scenario, bubble sort can be quite efficient, with a time complexity of O(n).

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

Answer:

Worst-case Big-O complexity ($O(n^2)$):

Picture a scenario where you have a list of numbers, and you want to sort them using selection sort. In the worst-case scenario, selection sort involves nested loops. It repeatedly scans the list to find the smallest (or largest) element and moves it to the beginning (or end) of the unsorted portion of the list. This process continues for each element in the list, resulting in a lot of comparisons and swaps. When you have n elements in the list, the number of comparisons and swaps can grow quadratically, leading to a time complexity of $O(n^2)$.

Best-case Big-O complexity ($O(n^2)$):

Even if the list is already sorted, selection sort still needs to scan through the entire list to confirm that it's sorted. This is because it's designed to find the smallest (or largest) element and move it to the appropriate position. So, even in the best-case scenario where the list is already sorted, selection sort still needs to make the same number of comparisons and swaps as in the worst case, resulting in a time complexity of $O(n^2)$.

3. Does selection sort require any additional storage (i.e. did you have to allocate any extra memory to perform the sort?) beyond the original array?

Answer:

Selection sort does not require any additional storage beyond the original array. It operates directly on the elements within the array and rearranges them in place.
Selection sort works by repeatedly finding the minimum (or maximum) element from the unsorted portion of the array and swapping it with the first (or last) unsorted element. This process continues until the entire array is sorted. However, because selection sort involves nested loops and does not take advantage of any pre-sortedness of the array, its performance remains quadratic in both the worst and best cases. This lack of efficiency is why the Big-O complexity does not change significantly based on the initial state of the array.

4. Would the Big-O complexity of any of these algorithms change if we used a linked list instead of an array?

Answer:
Yes, the Big-O complexity of selection sort would change if we used a linked list instead of an array.
In an array, selection sort involves directly accessing elements by index, which allows for constant-time access to any element. However, in a linked list, accessing elements is not as efficient because we have to traverse the list sequentially from the beginning to reach a specific element. This change in access pattern affects the performance of selection sort.

5. Explain what you think Big-O complexity of sorting algorithm that is built into the C libraries is. Why do you think that?

Answer:
The sorting algorithm typically built into C libraries, such as qsort() in stdlib.h, is often a variation of quicksort or mergesort. These algorithms are chosen because they have good average-case time complexity and are efficient for sorting large datasets.

Quicksort and mergesort are efficient sorting algorithms that divide the array into smaller subarrays, sort them recursively, and then combine them in a sorted manner. These algorithms typically have good performance characteristics and are widely used in practice. Their average-case time complexity of O(nlogn) makes them suitable for sorting large datasets efficiently. The choice of these algorithms in C libraries reflects their effectiveness in real-world applications.

Selection sort involves repeatedly finding the minimum (or maximum) element from the unsorted portion of the array and swapping it with the appropriate element. However, because it involves nested loops and does not take advantage of any pre-sortedness of the array, its performance remains quadratic in both the worst and best cases. This lack of efficiency is why the Big-O complexity doesn't change significantly based on the initial state of the array.