LAB 01 - REPORT

CO322

W.W.K. JAYATHUNGA
E/19/166

How does the performance vary with the input size?

The performance of sorting algorithms such as bubble sort, selection sort, and insertion sort can vary significantly with the size of the input.

1. Bubble Sort:

   Bubble sort has a worst-case and average time complexity of $O(n^2)$, where n is the size of the input list. This means that if we double the size of the list, the time it takes to sort might quadruple. This is because bubble sort compares each pair of adjacent items and swaps them if they are in the wrong order, leading to $n-1$ passes through the list.

2. Selection Sort:

   Like bubble sort, selection sort also has a worst-case and average time complexity of $O(n^2)$. This is because it divides the input into a sorted and an unsorted region, and repeatedly selects the smallest (or largest) element from the unsorted region and moves it to the sorted region. Despite this, selection sort performs fewer swaps compared to bubble sort, so it may be faster in scenarios where write operations are costly.

3. Insertion Sort:

   Insertion sort has a worst-case time complexity of $O(n^2)$, but its best-case time complexity is O(n), which occurs when the input list is already sorted. This is because insertion sort scans the unsorted portion of the list and inserts each element into its correct position in the sorted portion of the list.

For example, we might observe that the time taken for these algorithms increases quadratically with the size of the input, consistent with their $O(n^2)$ time complexity. However, we might also notice differences between the algorithms due to factors not captured by the big O notation, such as the number of swaps performed

Do the empirical results you get agree with theoretical analysis?

The empirical results obtained from the implementation of sorting algorithms should ideally align with the theoretical analysis. However, this might not always be the case due to various factors:

1. Theoretical Analysis:

   Theoretical analysis provides an estimate of the performance of an algorithm based on its time and space complexity. For example, bubble sort, selection sort, and insertion sort all have a worst-case time complexity of $O(n^2)$, which means the time taken by these algorithms increases quadratically with the size of the input.

2. Empirical Results:

   Empirical results are obtained by actually running the implemented algorithms on a computer. These results can vary based on factors like system hardware, software, programming language used, and even the specific dataset used for testing. In our task, we're implementing and testing bubble sort, selection sort, and insertion sort. If our empirical results do not align with the theoretical analysis, it could be due to reasons like:

- Optimizations in the Code:

  Certain optimizations in our code can make the algorithm perform better than its theoretical time complexity. For example, an optimized version of bubble sort stops early if it finds the array is already sorted.

- System Factors:

Factors like CPU speed, memory, and system load can affect the running time of our algorithms.

- Programming Language:

  The efficiency of the programming language and its execution environment can also impact the empirical results.

Therefore, while the empirical results should ideally agree with the theoretical analysis, discrepancies can occur due to the above factors.

How did/should you test your code. Explain the test cases you used and the rationale for use them.

To test the sorting algorithms (bubble sort, selection sort, and insertion sort), we should use a variety of test cases to ensure the correctness and efficiency of our code.

1. Random Test Cases:

   Generate arrays with random numbers. This helps in testing the average-case performance of the algorithms1.

2. Sorted Test Cases:

   Use already sorted arrays. This can test the best-case performance of the algorithms, especially for algorithms like insertion sort that can perform better on sorted input.

3. Reversed Test Cases:

   Use arrays sorted in descending order. This often represents the worst-case scenario for many sorting algorithms.

4. Duplicate Elements:

Include duplicate elements in your arrays. This can test the stability of the sorting algorithms.

5. Large and Small Arrays:

   Test with both large and small arrays. This helps in understanding how the algorithms perform with varying input sizes.

The rationale for using these test cases is to ensure that our sorting algorithms work correctly and efficiently in different scenarios. It's important to note that the performance of sorting algorithms can greatly depend on the initial order of the input.