TONY LE HUYNH - GCAB012201

Q1.

The two sorting algorithms chosen for discussion are Bubble Sort and Merge Sort.

Bubble Sort is a simplistic sorting algorithm where steps are executed to sort through an array by comparing adjacent elements to each other and swapping them if they are in the incorrect order (Reference 1). Just like how bubbles rise in water, the Bubble Sort algorithm is characterised by how smaller or larger values “bubble” up to the top as the algorithm sorts through an array (Reference 2).

Below is a simple Bubble Sort algorithm pseudo code example (Reference 1):

```
def bubble_sort(arr):
    # Find the length of the array

    # Use a For Loop to iterate over the array from the first element to the
second last element

        # Use another For Loop (nested) to iterate over the array from the first
element to the element n-i-2

            # If the current element is greater than the next element then swap
the elements
```

In the pseudo code example, the Bubble Sort function takes an array as the input. The algorithm then works by comparing consecutive items in the array. If the items are out of place, they are then swapped (Reference 2). With each iteration of the algorithm, as shown by the nested loops, the highest value will “bubble” up to the right or end of the array, thus resulting in the array being sorted in ascending order.

The time complexity of Bubble Sort algorithms is O(n^2). This is due to the fact where in the worst case scenario, the list or array is reverse sorted (decreasing order), which results in the Bubble Sort Algorithm performing a total of n(n-1)/2 comparisons and n(n-1)/2 swaps (Reference 1). Therefore the worst case scenario and average case scenarios for Bubble Sort algorithms have a time complexity of O(n^2), which is also highlighted by the nested loops used in the algorithm.

However, there is a best case scenario where the array is already sorted, meaning the Bubble Sort algorithm only performs a total of n-1 comparisons as adjacent elements in the array are compared with each pass with no swaps taking place (Reference 1). This means in this best case scenario, Bubble Sort has a time complexity of O(n).

Another sorting algorithm is Merge Sort, which can be identified by its use of a divide-and-conquer approach for sorting an array. Merge Sort is also a recursive algorithm, where recursion is a process where a function calls itself to solve a smaller sub-problem (Reference 3).

Merge Sort works by continuously splitting an array in half into smaller subarrays. These subarrays are recursively sorted and then merged back together to produce the final fully sorted array (Reference 1). These

steps are shown in the below pseudo code example of a Merge Sort algorithm:

```
def merge_sort(arr):
    # Divide the input array into two halves

    # Recursively sort each half

    # Merge the sorted halves into a sorted whole

def merge(left, right):
    # Initialise an empty list to hold the merged result

    # Initialise two pointers i and j to the first element of the left and right
sub-arrays

    # While LOOP to compare consecutive elements and append based on ascending
order

    # Add any remaining elements in the left or right sub-arrays to the result
list

    # Return the merged result
```

For Merge Sort, the time complexity for best, average and worst case scenarios is O(n log n), making it relatively efficient for sorting large arrays. This is due to the fact that the algorithm always splits the array in half and takes linear time to merge the two halves, meaning that the number of comparisons and swaps grow logarithmically with the size of the input array (Reference 1).

This means that Merge Sort is a more efficient and better performing algorithm than Bubble Sort, as it has a lower time complexity of O(n log n) for a worst-case scenario as compared to Bubble Sort's time complexity of O(n^2). However, if the array is already sorted then Bubble Sort will out perform Merge Sort as Bubble Sort has a time complexity of O(n) in the best-case scenario where the input array is already sorted. Merge Sort will still go through its whole process even if the array is already sorted, meaning it will be slower than Bubble Sort in this specific case. For large data sets, Bubble Sort performs slowly compared to Merge Sort.

It is also worth to consider some other cases where Bubble Sort is more suitable than Merge Sort. The advantages of Bubble Sort are that it is simple to implement and does not require any additional memory space. On the other hand, Merge Sort requires more code to implement due to the dividing, sorting and merging of the subarrays and also requires additional memory used to store the subarrays during the sorting process (Reference 1). This means that for very small input array sizes, Bubble Sort can sometimes be faster than Merge Sort. Overall, Merge Sort is generally more efficient than Bubble Sort in terms of performance.

Q2.

Two types of search algorithms include - Linear Search and Binary Search.

Linear Search is a search algorithm that checks through every element of an array until the desired element is identified (Reference 4). Below is a pseudo code example of a Linear Search algorithm:

```
def linear_search(arr, x):
    # Iterate through all elements in the array using a for loop
        # If the current element is equal to the search value, return the index
```

It can be seen from this example that Linear Search is very simple in nature, where each iteration of the algorithm checks sequentially through each element of the array one at a time until the desired element is identified. An advantage of Linear Search is that it can be used on both sorted and unsorted arrays (Reference 1).

Thus, Linear Search has a time complexity of O(n), with n representing the number of elements in the array. The algorithm's performance is linearly correlated with the number of elements in the array being searched. In the worst case scenario, the desired element is not present in the array meaning that the algorithms will have had to iterate through all the elements in the array - showing how the time complexity is O(n) (Reference 1). In the best case scenario, the desired element is at the start of the array which results in a time complexity of O(1).

Binary Search is also a type of search algorithm used for identifying a desired element's position in a sorted array. Here is a pseudo code example of a Binary Search algorithm:

```
def binary_search(arr, x):
    # Initialise search points for the start and end of the search range

    # Use While Loop
        # Find the middle index of the search range

        # If the middle element is equal to the search value, return the index

        # If the middle element is greater than the search value, search the left
half of the range

        # If the middle element is less than the search value, search the right
half of the range
```

Binary Search relies on the array already being sorted. From the pseudo code, it can be seen that how Binary Search works is by first initialising the search range of the algorithm as the start and end of the array. It then splits the search interval in half. If the middle index of the array matches the desired element being searched, the search algorithm finishes. Based on if the middle element of the array is greater or less than the desired element for search, the Binary Search algorithm will then search either the left half or the right half of the array. This process is repeated until the desired element is identified in the array (Reference 1).

In the worst case scenario, Binary Search has a time complexity of O(log n), shown by how Binary Search splits the search interval by half with each iteration. The best case scenario would be if the middle element of the array is identified as the desired element, resulting in a time complexity of O(1) (Reference 5).

It can be seen that Binary Search, with a time complexity of O(log n), performs better and is more efficient than Linear Search which has a time complexity of O(n). The drawback of Linear Search is that the algorithm is slow for large datasets with the time that it takes lineary being correlated with the number of elements in the

array. On the other hand, Binary Search is much better suited for large arrays and would process faster than Linear Search (Reference 1).

There are some drawbacks to Binary Search. Firstly, Binary Search requires the array to already be sorted and ordered. If the array isn't already sorted, then an additional step will need to be added to sort the array which can result in an additional O(n log n) time complexity for this sorting step - making Binary Search less efficient in this case for small arrays (Reference 1). Also, elements of the array must be able to be ordered, with the ordering well-defined, otherwise the Binary Search algorithm will not work as intended.

In the above scenarios representing the drawbacks of Binary Search, Linear Search would be more efficient. This is due to the advantages of Linear Search where it is simple to implement, and can be used on unordered and unsorted arrays, regardless of the types of data contained in the array (Reference 1). For very small arrays, Linear Search is well suited for use.

For large datasets that are already sorted however, Binary Search performs faster than Linear Search and is therefore the more viable option in this case.

REFERENCES

(1) GeeksforGeeks

(2) freeCodeCamp

(3) learnlearn

(4) Programiz

(5) JavaTPoint