Name: Goudham Suresh Student GUID: 2563586s

Practical Algorithms AE1

Problem 1

(a)

See Code Within “solution_part_1_2.py”

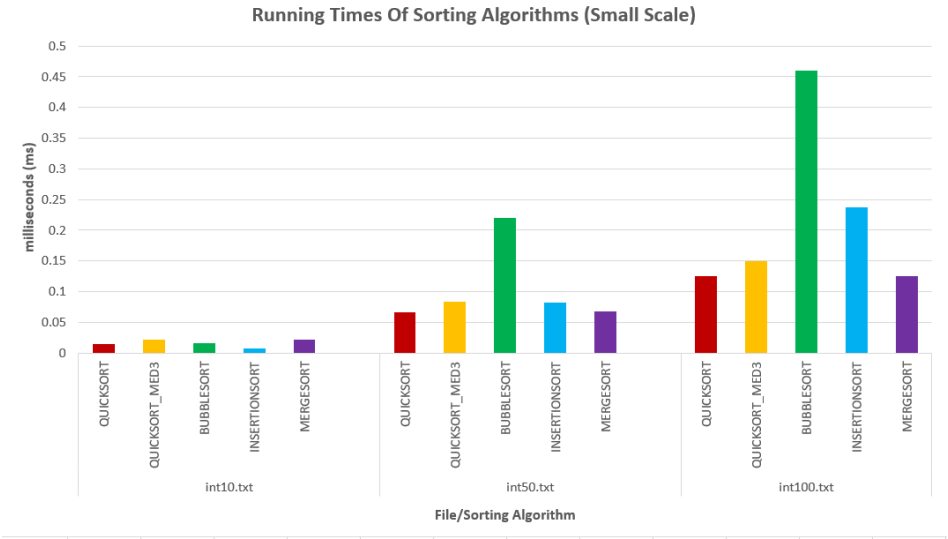
(b)

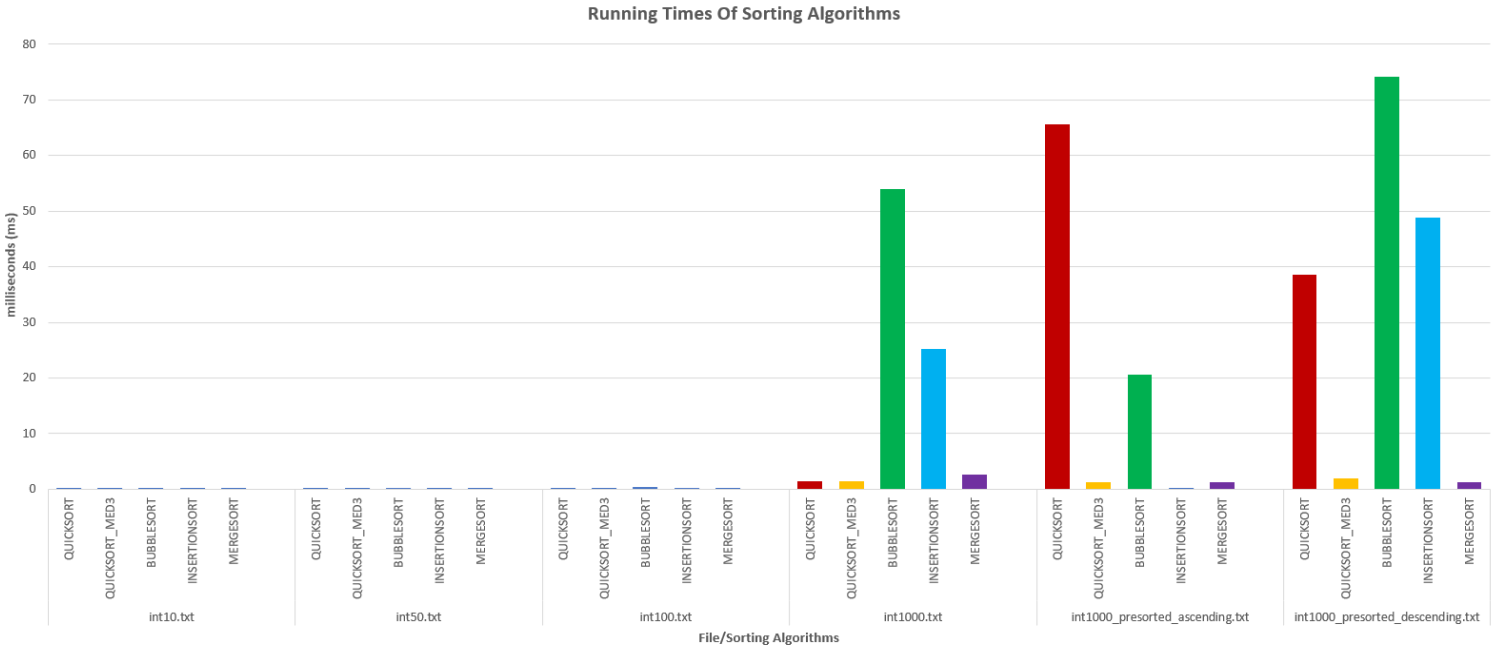
See Code Within “solution_part_1_2.py”

Problem 2

See Code Within “solution_part_1_2.py”

Column Charts





Analysis

2 Column Charts have been included to showcase the main reason for testing algorithms on large datasets instead of small ones. For datasets of **10**, **50** and **100** integers, we can see large variations on the small-scale column chart. However, on the bigger column chart. We can see that the differences between sorts at the smaller scale are negligible at best.

Focusing on datasets of **1000**, **1000_sorted_ascending** and **1000_sorted_descending**, Quicksort/Quicksort_Med3 and Merge sorts use a Divide-And-Conquer approach that results in a Big O(n log n) which allows for faster sorting speeds than Bubble and Insertion sort which have a Big O(n^2). Furthermore, it should be noted that Quicksort takes extremely long compared to other sorts when data is sorted ascending, this is due to the partition always being unbalanced and resulting in Big O(n) running time. The Insertion Sort within **1000_sorted_ascending** is incredibly fast, the reason for this being that it is already sorted in ascending order.

Main Conclusions to Draw:

- At a small scale, our results do not show an accurate and true representation of how the algorithm scales up when provided with larger datasets.
- The benefit of the Quicksort Divide-And-Conquer approach is lost when data is sorted in such a way that results in unbalanced partitions.

Problem 3

(a)

See Code Within “solution_part_3_4.py”

(b)

See Code Within “solution_part_3_4.py”

(c)

Dynamic Set Linked List

- ADD() -> This function first calls IS-ELEMENT() to check if the element already exists within the set or not. If the element does not exist, the new element to add is inserted into the head of the LinkedList. Due to the traversal within IS-ELEMENT(), this function has a time complexity of **O(n).**
- REMOVE() -> This function first calls a helper function called RETRIEVE-ELEMENT() which checks if the element already exists within the set or not. The node containing the element is returned or None if the element does not exist. The node is then deleted by updating the pointers of the previous and next nodes. Due to the traversal within RETRIEVE-ELEMENT(), this function has a time complexity of **O(n).**
- IS-ELEMENT() -> This function iterates through all the nodes within the Backing LinkedList and returns True if the element has been found. False is returned if all nodes have been iterated through as this means the element does not exist. Due to the traversal of all potentially all nodes, this function has a time complexity of **O(n)**
- SET-EMPTY() -> This function simply checks if the head node is None or not. True is returned if the head is None, if not then False is returned. This function runs in constant time **O(1)** as the running time does not depend on the size of the backing LinkedList.
- SET-SIZE() -> This function traverses through the LinkedList and adds 1 to a counter as each node is encountered. The counter is then returned. Due to the traversal of the backing LinkedList, this function has a time complexity of **O(n).**

Dynamic Set Array

- ADD() -> This function first calls the IS-ELEMENT() method to check if the element already exists within the backing Array. If the element does not exist AND the set has space for more elements, the currentSize counter is incremented and the element is then inserted at the tail. Due to the traversal of IS-ELEMENT(), the time complexity of this function is **O(n)**
- REMOVE() -> This function uses a helper function GET-ELEMENT-POSITION() which returns the position of the element within the backing array if it exists. None is returned if the element does not exist. If the element does exist at the last position of the array, it is set to 0. If it's in the middle of the backing array, a for loop is used to iterate from the position of the element to be deleted to shift the elements to ensure that backing array is backfilled properly. This function has a time complexity of **O(n)** as the array is traversed multiple times in different parts of the code.
- IS-ELEMENT() -> This function traverses the backing Array with a for loop and if the element exists at the i'th index then True is returned. If the entire backing array has been traversed, False is returned as it means the element does not exist. Due to the traversal of the backing array, this function has a time complexity of **O(n)**.
- SET-EMPTY() -> This function simply checks if the currentSize variable within the set is equal to -1 or not. True is returned if it does equal to -1 and False if not. This function runs in constant time **O(1)** as the running time of this function is not influenced by the number of elements within the set.
- SET-SIZE() -> This function simply returns the value of currentSize + 1. The one has been added as the currentSize variable is zero-indexed and represents the number of elements within the set - 1. For the same reasons given in SET-EMPTY(), this function runs in constant time **O(1).**

(d)

- One disadvantage of the LinkedList approach is that each node occupies more space within the memory as opposed to the Array implementation. However, the LinkedList approach is more efficient with its use of memory as it's dynamic and only takes up as much as it's needed whereas the Array approach needs a fixed size of memory to be allocated beforehand.
- One advantage of the Array approach is that the memory allocated is of a fixed size so there is no risk of potentially running out of memory for elements. On the other hand, with a dynamic data structure such as a LinkedList, the possibility of memory running out is very real and can happen.
- Within my Array-based implementation, the deletion of an element is particularly inefficient as part of the array will need to get rewritten to be backfilled properly. However, within a doubly linked list, the removal of an element simply involves updating the pointers of the previous and next node.
- Within my LinkedList-based implementation, retrieving the size of the set is inefficient/disadvantageous as I am traversing the entire list and incrementing a size counter every time a node is encountered. This results in a time complexity of **O(n)** whereas the time complexity for my Array-based implementation is **O(1)** by having a size counter tied to the ADT itself. The SET-SIZE() method can be shaved down to constant time if a size counter was implemented and incremented/decremented whenever an element was added/removed from the Set.

(e)

IS_ELEMENT()

If the elements within the LinkedList were sorted, I believe the worst-case scenario for the IS-ELEMENT() method would still be **O(n)**. However, performance could be improved by implementing an if statement checking if the element to insert is less than the current value at the head node, which then would result in a time complexity of **O(1)**. The traversal of the elements would happen if the previous if statement evaluated to False. To conclude, the Big O (worst case) time complexity would remain **O(n).**

ADD()

If the elements within the LinkedList were sorted, I believe the worst-case scenario for the ADD() method would still be **O(n)**. Usually, the insertion of elements for a LinkedList would be in constant time as the elements would be inserted into the head. However, as this is a dynamic set, the IS-ELEMENT() method is called which in the worst-case scenario will be **O(n)**. To insert an element in a sorted fashion, the list would have to be traversed until an element greater than the element to insert has been found and insert a new node before the greater element. In the worst case, the traversal might go until the end of the list which would result in a time complexity of **O(n).**

Problem 4

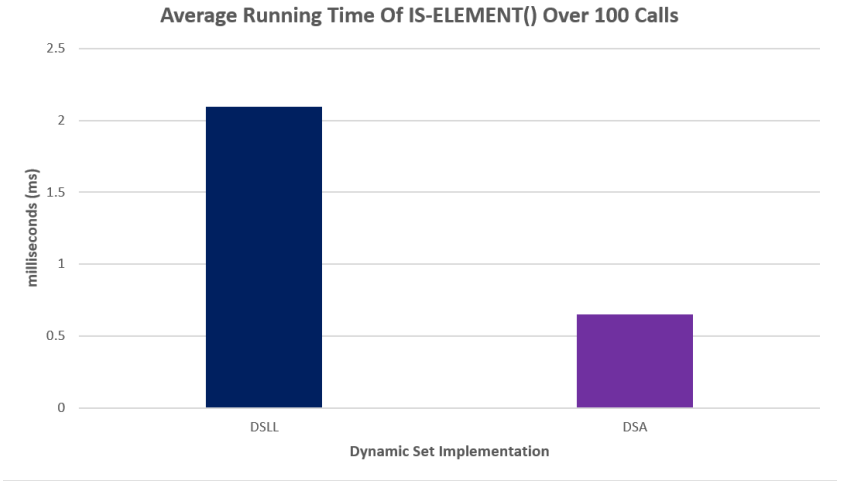
(a)

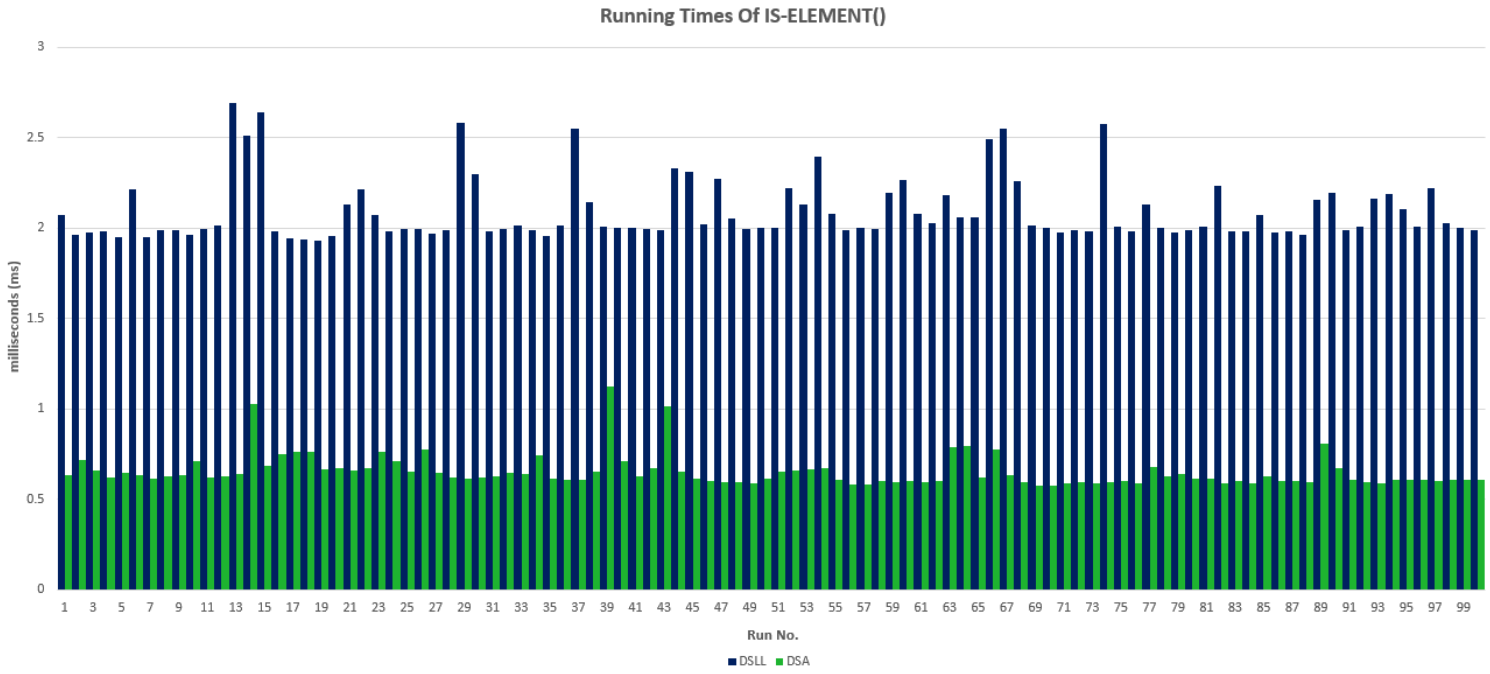
View “results_problem4.txt”
See Method “empirical_study()” Within “solution_part_3_4.py” For Average Running Times

(b)

(Analysis On Next Page)

Name: Goudham Suresh Student GUID: 2563586s





The first column chart represents the average running times over 100 calls for both implementations, the second column chart shows the individual runs and the time that it took.

When looking at both charts, it is abundantly clear that the Array-based implementation is a lot faster than the LinkedList-based implementation. The average time over 100 calls for DSLL was **2.09ms** compared to DSA which has an average time of **0.65ms** which results in a **320%** increase in running time when comparing DSA to DSLL.

The disparity between speeds can be explained through how Arrays and LinkedList's are allocated memory. An array is stored and allocated as a contiguous block of memory and therefore allows for very easy access to all the elements with an offset from the beginning. On the other hand, LinkedList use nodes which have pointers to other nodes which can potentially be in different places in memory. While the Python Programming language will have some clever code to try and mitigate this, while traversing the LinkedList, the language will have to calculate where the node is from the memory location and then read the value which is much costlier than the Array-based implementation.