Trevor Swan **CSDS233 ASSIGNMENT 5**

Part 1) Sorting Algorithms

Q1 Merge, Quick and Selection Sort (12 points)
Given the following list {12,5,9,3,15,7,2,10,6,8} and sorting functions below, write out each step of the sorting and comparison process until the list is fully sorted.

a) Merge Sort (4pts)
b) Quick Sort (4pts)
c) Selection Sort (4pts)

a) Merge Sort on {12, 5, 9, 3, 15, 7, 2, 10, 6, 8}

(i) Split list into two halves

Left Half: {12, 5, 9, 3, 15}

Right Half: {7, 2, 10, 6, 8}

(ii) Split the left half in two

Left Half: {12, 5}

Right Half: {9, 3, 15}

(iii) Split the left half and sort

=> {12} and {5}

sort to: {5, 12} from 12>5

(iv) Split right half in two

Left Half: {9}

Right Half: {3, 15}

(v) Split right half in two then sort

=> {3} and {15}

Sort to: {3, 15} from 3<15

(vi) Merge {9} and {3, 15} sorted

{3, 9, 15} from 9>3

(vii) Merge {5, 12} and {3, 9, 15} in sorted order

{3, 5, 9, 15} from 5 > 3, 5<9

{3, 5, 9, 12, 15} from 12 > 3, 12>5, 12>9

(viii) Split the right half in two

left Half: {7, 2}

Right Half: {10, 6, 8}

(ix) Split the left half and sort

=> {7} and {2}

Sort to: {2, 7} from 7>2

(x) Split right half in two

Left Half: {10}

Right Half: {6, 8}

(xi) Split right half in two

=> {6} and {8}

Sort to: {6, 8} from 6<8

(xii) Merge {10} and {6, 8} sorted

{6, 8, 10} from 10>6, 10>8

(xiii) Merge {2, 7} and {6, 8, 10} in sorted order

{2, 6, 8, 10} from 2<6

{2, 6, 7, 8, 10} from 7>2, 7>6, 7<8

(xiv) Merge {3, 5, 9, 12, 15} and {2, 6, 7, 8, 10} in sorted order

{2, 3, 5, 9, 12, 15} from 2<3

{2, 3, 5, 6, 9, 12, 15} from 6>2, 3, 5; 6<9

{2, 3, 5, 6, 7, 9, 12, 15} from 7>2, 3, 5, 6; 7<9

{2, 3, 5, 6, 7, 8, 9, 12, 15} from 8>2, 3, 5, 6, 7; 8<9

{2, 3, 5, 6, 7, 8, 9, 10, 12, 15}

from 10 > 2, 3, 5, 6, 7, 8, 9; 10<12

b) Quick Sort on {12, 5, 9, 3, 15, 7, 2, 10, 6, 8}

(i) Choose a Pivot: 8

(ii) Partitioning:

(ii.a) $12 > 8 \rightarrow$ no action

(ii.b) $5 < 8 \xrightarrow[5\&12]{\text{Swap}}$ {5, 12, 9, 3, 15, 7, 2, 10, 6, 8}

(ii.c) $9 > 8 \rightarrow$ no action

(ii.d) $3 < 8 \xrightarrow[12\&3]{\text{Swap}}$ {5, 3, 9, 12, 15, 7, 2, 10, 6, 8}

(ii.e) $15 > 8 \rightarrow$ no action

(ii.f) $7 < 8 \xrightarrow[9\&7]{\text{Swap}}$ {5, 3, 7, 12, 15, 9, 2, 10, 6, 8}

(ii.g) $2 < 8 \xrightarrow[12\&2]{\text{Swap}}$ {5, 3, 7, 2, 15, 9, 12, 10, 6, 8}

(ii.h) $10 > 8 \rightarrow$ no action

(ii.i) $6 < 8 \xrightarrow[15\&6]{\text{Swap}}$ {5, 3, 7, 2, 6, 9, 12, 10, 15, 8}

(ii.j) Place 8 correctly $\rightarrow$ {5, 3, 7, 2, 6, 8, 12, 10, 15, 9}

Pivot 8 is now placed at the correct position

(iii) Recursively Sort leftsublist {5, 3, 7, 2, 6}; choose pivot: 6

(iv) Partitioning:

(iv.a) $5 < 6 \xrightarrow[5\&5]{\text{Swap}}$ {5, 3, 7, 2, 6}

(iv.b) $3 < 6 \xrightarrow[3\&5]{\text{Swap}}$ {3, 5, 7, 2, 6}

(iv.c) $7 > 6 \rightarrow$ no action

(iv.d) $2 < 6 \xrightarrow[7\&2]{\text{Swap}}$ {5, 3, 2, 7, 6}

(iv.e) Place 6 correctly: {5, 3, 2, 6, 7}

↖ right sublist

(v) Recursively Sort left sublist {5, 3, 2} with Pivot: 2

(v.a) $5 > 2 \rightarrow$ no action

(v.b) $3 > 2 \rightarrow$ no action

(v.c) Swap 2 and 5, as 5 is first element greater than pivot $\longrightarrow$ {2, 3, 5}

(vi) Combine the sorted sublists: {2, 3, 5, 6, 7}

(vii) Recursivly Sort right sublist {12, 10, 15, 9} with Pivot: 9

(vii.a) $12 > 9 \rightarrow$ no action

(vii.b) $10 > 9 \rightarrow$ no action

(vii.c) $15 > 9 \rightarrow$ no action

(vii.d) Place Pivot correctly: {9, 10, 15, 12}

(viii) Recursively Sort right sublist {10, 15, 12} with Pivot 12

(viii.a) $10 < 12 \xrightarrow[10\&10]{\text{Swap}}$ {10, 15, 12}

(viii.b) $15 > 12 \rightarrow$ no change

(viii.c) Place Pivot correctly: {10, 12, 15}

(ix) Combine sublists to get {9, 10, 12, 15}

(x) Combine the two sorted sublists: {2, 3, 5, 6, 7, 8, 9, 10, 12, 15}

c) Selection Sort on {12, 5, 9, 3, 15, 7, 2, 10, 6, 8}

(i) Find minimum element and swap with first element

Minimum: 2 => Swap with 12 {2, 5, 9, 3, 15, 7, 12, 10, 6, 8}

(ii) Find minimum element and swap with next element

Minimum: 3 => Swap with 5 {2, 3, 9, 5, 15, 7, 12, 10, 6, 8}

(iii) Find minimum element and swap with next element (unsorted sublist)

Minimum: 5 => Swap with 9 {2, 3, 5, 9, 15, 7, 12, 10, 6, 8}

(iv) Find minimum element and swap with next element (unsorted sublist)

Minimum: 6 => Swap with 9 {2, 3, 5, 6, 15, 7, 12, 10, 9, 8}

(v) Find minimum element and swap with next element (unsorted sublist)

Minimum: 7 => Swap with 15 {2, 3, 5, 6, 7, 15, 12, 10, 9, 8}

(vi) Find minimum element and swap with next element (unsorted sublist)

Minimum: 8 => Swap with 15 {2, 3, 5, 6, 7, 8, 12, 10, 9, 15}

(vii) Find minimum element and swap with next element (unsorted sublist)

Minimum: 9 => Swap with 12 {2, 3, 5, 6, 7, 8, 9, 10, 12, 15}

(viii) Find minimum element and swap with next element (unsorted sublist)

Minimum: 10 => Swap with 10 → no change

(ix) Find minimum element and swap with next element (unsorted sublist)

Minimum: 12 => Swap with 12 → no change

Array is fully sorted: {2, 3, 5, 6, 7, 8, 9, 10, 12, 15}

Q2 Bubble Sort (12 points)

a) Implement the **bubble sort** algorithm to sort the following list of integers in ascending order {34, 7, 23, 32, 5, 62}. Provide a step-by-step explanation of each pass through the list, detailing the comparisons and swaps made. (6pts)

Swap outof place & adjcent items until sorded fully

Pass 1 {34, 7, 23, 32, 5, 62}

(i) $34>7$ so swap → {7, 34, 23, 32, 5, 62}

(ii) $34>23$ so swap → {7, 23, 34, 32, 5, 62}

(iii) $34>32$ so swap → {7, 23, 32, 34, 5, 62}

(iv) $34>5$ so swap → {7, 23, 32, 5, 34, 62}

(v) $34<62$ so no change needed

Pass 2 {7, 23, 32, 5, 34, 62}

(i) $7<23$ so no change

(ii) $23<32$ so no change

(iii) $32>5$ so swap → {7, 23, 5, 32, 34, 62}

(iv) $32<34$ so no change

(v) $34<62$ so no change

Pass 3 {7, 23, 5, 32, 34, 62}

(i) $7<23$ so no change

(ii) $23<5$ so swap → {7, 5, 23, 32, 34, 62}

(iii) $23<32$ so no change

(iv) $32<34$ so no change

(v) $34<62$ so no change

Pass 4 {7, 5, 23, 32, 34, 62}

(i) $7<5$ so swap → {5, 7, 23, 32, 34, 62}

(ii) $7<23$ so no change

(iii) $23<32$ so no change

(iv) $32<34$ so no change

(v) $34<62$ so no change

Pass 5 {5, 7, 23, 32, 34, 62}

(i) $5<7$ so no change

(ii) $7<23$ so no change

(iii) $23<32$ so no change

(iv) $32<34$ so no change

(v) $34<62$ so no change

There was no changes made on pass 5, so the algorithm completes!

Final Sorted List: {5, 7, 23, 32, 34, 62}

b) Analyze the time complexity of bubble sort in the best case, worst case and average case scenarios. Discuss how the initial order of elements affects the performance of the algorithm. (6pts, 2pts for each scenario : Total 6pts)

Best Case:
This means that the list is already in sorted order. In this case, the algorithm only compares elements but doesn't perform any swaps. In a good implementation, the inner loop of the algorithm will acknowledge that there were no swaps made, and the algorithm will terminate early after one pass. Here, there are O(0) swaps, and there are n-1 comparisons made, so the algorithm runs on O(n) time. The overall time complexity for this is O(n).

Worst Case:
This means that the list is sorted in reverse order. In this case, the algorithm must swap every pair of adjacent elements. It will require n-1 swaps, and requires multiple swaps to bubble up the next element to its correct position. The inner loop will have to perform n-1, n-2, n-3, …, 1 total comparisons across a total of n-1 passes, so there are O(n^2) total comparisons. There are also going to have to be O(n^2) total swaps as each pass must swap adjacent elements. This results in an overall time complexity of O(n^2).

Average Case:
This means that the list is in random order. In this case, it can be assumed that about half of the elements are in the wrong position. The number of comparisons and swaps is somewhere in between the best and worse cases. While fewer swaps and comparisons will be made, the complexity will remain closer to the worst case scenario. In this case, Comparisons, Swaps, and the overall time complexity should be O(n^2).

Effect of Initial Order on Performance:
1. In a sorted list, bubble sort terminates early and the performance is efficient O(n).
2. In reverse order, every element must be moved to its correct position, maximizing the number of comparisons and swaps, resulting in poor performance O(n^2).
3. In random order, the algorithm behaves unpredictably as it must make multiple passes with a mix of total number of swaps and comparisons. The time complexity of the algorithm remains quadratic O(n^2).

Q3 Bucket Sort (14 points)

a) You are given the following list of floating-point numbers: {0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68}. Use the bucket sort algorithm to sort the list. Please show each step for full credit, including the contexts of each bucket before and after sorting. Use 5 equal-sized buckets for this. (4pts)

Create 5 equal-sized buckets, distribute elements

$B_1 = [0, 0.2)$	Add	0.17, 0.12
$B_2 = [0.2, 0.4)$	Add	0.39, 0.26, 0.21, 0.23
$B_3 = [0.4, 0.6)$	Add	
$B_4 = [0.6, 0.8)$	Add	0.78, 0.72, 0.68
$B_5 = [0.8, 1.0)$	Add	0.94

Buckets before Sorting	$\rightsquigarrow$	Buckets after Sorting
$B_1 = \{0.17, 0.12\}$		$B_1 = \{0.12, 0.17\}$
$B_2 = \{0.39, 0.26, 0.21, 0.23\}$		$B_2 = \{0.21, 0.23, 0.26, 0.39\}$
$B_3 = \{\}$		$B_3 = \{\}$
$B_4 = \{0.78, 0.72, 0.68\}$		$B_4 = \{0.68, 0.72, 0.78\}$
$B_5 = \{0.94\}$		$B_5 = \{0.94\}$

Group the buckets into one list from B_1 to B_5

$\{0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94\}$

b) How does the number of buckets affect the performance and accuracy of the bucket sort algorithm? (3pts)

If there are not enough buckets, then the algorithm becomes less efficient as the buckets will contain more elements, leading to more time spent on sorting individual buckets, reducing performance. Fewer buckets require more elements per bucket, and the time complexity will tend towards O(nlogn), where it could be O(n) in the best case. In terms of accuracy, there can be a slight reduction in accuracy as elements can be too widely varied in the same bucket, which reduces the effectiveness of the algorithm. This is generally a small impact, however. If there are too many buckets, then the algorithm becomes less efficient as there are fewer elements per bucket. This creates more memory overhead, as there is a high likelihood of empty or 1-element buckets. The largest deficiency that arises with more buckets is the memory overhead, decreasing performance. In terms of accuracy, there should be no impact on using too many buckets. With two many buckets, there will be few elements per bucket, with some even having 0, or 1 element. Here, there is no sorting to be done on these buckets, and the algorithm simply iterates over the input, which results in no accuracy impact.

c) Given the non-uniform distribution of the dataset, how does this impact the effectiveness of the Bucket sort algorithm? How can we modify the algorithm to handle the uneven distribution? (3pts)

With a non-uniform dataset, the effectiveness is significantly reduced as there will be some buckets with extremely large amounts of elements and others with none. This bring the time complexity up, as well as the space complexity as there is a large amount of buckets with very few elements. There are a few things we could to do to handle this type of data:

1. Dynamic Bucket Allocations: We could define buckets based off of the statistics of the data, such as the mean or median so that there can be a more balanced distribution of data.
2. Increasing the Number of Buckets: If even more buckets are used, this will reduce the number of buckets with overflowed elements with the downside of increasing the overhead of managing too many buckets.
3. Weighted Bucket Ranges: Instead of using a set increment for the bucket sizes, we could use some with more possible data points, and others with fewer. This reduces the amount of buckets with to few elements, and also prevents other buckets from getting to large.
4. Reprocess the Data: To better distribute the data, we could normalize the data via exponentiation or logarithmic manipulation such that the values are more evenly distributed.

d) Compare the time complexity of Bucket Sort with that of Quick Sort and Merge Sort. Under what circumstances would Bucket Sort be more efficient as compared to the other two sorting algorithms? List two.(4pts)

Algorithm	Best Case	Average Case	Worst Case
Bucket Sort	$O(n+K)$	$O(n+K)$	$O(n^2)$ bad bucket distribution
Quick Sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$ poor pivot choice
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$

where 'n' is number of total elements, and 'K' is number of bucket

1. Bucket Sort: Performs better with uniformly distributed data, and sorting within each bucket is typically a much smaller operation, reducing the cost to nearly linear complexity.
2. Quick Sort: Performs very reliably unless a pivot is chosen incorrectly. This results in quadratic complexity.
3. Merge Sort: There is no bias to certain data sets or choices in merge sort, but it requires more memory than quick sort due to merging sub lists.

Bucket Sort can be most efficient (compared to Quick Sort and Merge Sort) on evenly or uniformly distributed ranges. It achieves near-linear time complexity. It can also pull through in scenarios were the data is dense and small with respect to its range. Here, we can arrange data into perfectly laid out buckets and the algorithm can shine.

Part 2) Insertion Sort on LinkedList

Q4

Implement the insertion sort algorithm to sort a singly linked list of integers in ascending order. Please provide a write-up of your approach, including pseudocode and analyze the time complexities of your solution.

Algorithm:

1. Initialize the algorithm
   A. Maintain a dummy node pointing to the head to allow for handling edge cases like inserting at the start of the list.
   B. Iterate through the original list one node at a time
2. Insert into Sorted List
   A. For each node in the unsorted part of the list, the correct position of each element should be found and placed in the correct portion of the sorted part of the list
3. Update Pointers
   A. Reset the pointers of the nodes to maintain the algorithms structure, and repeat until there are no more elements left in the unsorted part of the list

Pseudocode:

```
func insertionSort(head):
  return if head is null or head.next is null
  // Dummy node
  dummy = new Node()
  dummy.next = head
  // Track current node and loop
  current = head
  while current.next is not null:
    if current's next node val < current's val:
      // Grab the node to be moved
      toInsert = current.next
      current.next = toInsert.next
      // Find correct position for toInsert
      position = dummy
      while position's next node val < toInsert's val:
          position = position.next
      // Insert the node
      toInsert.next = position.next
      position.next = to Insert
    else:
      current = current.next
```

Time Complexity:

Each node is visited exactly once, contributing to O(n). T find the correct position, the algorithm traverses part of the sorted portion, resulting to a worst case time complexity of O(n^2) for the entire list. Thus, the time complexity of this implementation is O(n^2) only because there is n nodes that need to be checked, effectively n times each. Since the algorithm is in place, there is no memory overhead and the space complexity is given as O(1).