UNIVERSITY OF GHANA

Course Code and Title: CPEN 204 Data Structures and Algorithms

Student ID: 10987644

ASSIGNMENT 1

Question 1

1. Searching Algorithm: Binary Search

2. Sorting Algorithm: Merge Sort

i. Describe each algorithm with examples:

Binary Search:

- Description: Binary search is a search algorithm that works on sorted arrays. It repeatedly divides the search space in half until the target element is found or the search space is empty. It compares the target element with the middle element of the array and narrows down the search to the left or right subarray accordingly.

- Example: Let's search for the element 15 in the sorted array [2, 5, 8, 12, 15, 21, 28, 30].

  - Step 1: The middle element is 12. Since 15 > 12, we search the right subarray [15, 21, 28, 30].

  - Step 2: The middle element is 21. Since 15 < 21, we search the left subarray [15].

  - Step 3: The middle element is 15, and we have found the target element.

Merge Sort:

- Description: Merge Sort is a divide-and-conquer sorting algorithm that divides the unsorted array into smaller subarrays until each subarray contains only one element. It then merges these sorted subarrays to produce the final sorted array.

- Example: Let's sort the array [38, 27, 43, 3, 9, 82, 10].

  - Step 1: Divide the array into two subarrays: [38, 27, 43] and [3, 9, 82, 10].

  - Step 2: Recursively divide the subarrays until each subarray contains only one element: [38], [27], [43], [3], [9], [82], [10].

  - Step 3: Merge the sorted subarrays back together:

    Merge([38], [27]) -> [27, 38]

    Merge([43], [3]) -> [3, 43]

    Merge([9], [82]) -> [9, 82]

    Merge([10]) -> [10]

  - Step 4: Merge the remaining subarrays:

    Merge([27, 38], [3, 43]) -> [3, 27, 38, 43]

Merge([9, 82], [10]) -> [9, 10, 82]

- Step 5: Merge the final subarrays:

Merge([3, 27, 38, 43], [9, 10, 82]) -> [3, 9, 10, 27, 38, 43, 82]

ii. Compare their performance:

Binary Search:

- Time Complexity: O(log n), where n is the number of elements in the array.
- Space Complexity: O(1) (iterative) or O(log n) (recursive) for the auxiliary space.
- Performance: Binary search is efficient for large sorted arrays, as it reduces the search space by half in each iteration. However, it requires a sorted array, and the overhead of maintaining a sorted array can be significant.

Merge Sort:

- Time Complexity: O(n log n), where n is the number of elements in the array.
- Space Complexity: O(n) for the auxiliary space used during merging.

- Performance: Merge Sort is efficient for large arrays as it divides the array into smaller subarrays, sorts them, and then merges them back together. Its time complexity remains consistent for both average and worst-case scenarios.

iii. Advantages and Disadvantages:

Binary Search:

- Advantages:
  - Efficient for large sorted arrays as it reduces the search space in each iteration.
  - Time complexity of O(log n) makes it faster than linear search (O(n)) for large datasets.
- Disadvantages:
  - Requires the array to be sorted, which can be an additional overhead.
  - Not suitable for unsorted or dynamically changing arrays.

Merge Sort:

- Advantages:
  - Efficient for large datasets, as the time complexity remains consistent at O(n log n).

- Stable sorting algorithm that maintains the relative order of equal elements.

- Disadvantages:

  - Requires additional memory space for merging subarrays, making it less memory-efficient for large arrays.

  - Not suitable for small datasets or datasets with a small number of elements, as the overhead of recursion can be significant.

Question 2

For sorting a list L consisting of a sorted list followed by a few "random" elements, the Merge Sort algorithm would be especially suitable for such a task.

Explanation:

Merge Sort is a divide-and-conquer sorting algorithm that divides the input list into smaller sublists, recursively sorts them, and then merges the sorted sublists back together to produce the final sorted output. This approach is well-suited for sorting lists with a mix of already sorted elements and unsorted elements for the following reasons:

1. Stable Sorting: Merge Sort is a stable sorting algorithm, meaning it preserves the relative order of equal elements. In a situation where the original sorted list has elements with the same values, Merge Sort ensures that their relative positions are maintained after sorting. This characteristic is particularly useful when dealing with datasets containing elements with equal values.

2. Predictable Time Complexity: Merge Sort has a consistent time complexity of O(n log n) regardless of the initial order of elements in the input list. While other sorting algorithms like Quick Sort may have a worst-case time complexity of O(n^2) for partially sorted lists, Merge Sort guarantees efficient performance with its time complexity.

3. Divide-and-Conquer Approach: Merge Sort's divide-and-conquer approach enables it to efficiently sort the mixed list by dividing the problem into smaller subproblems. The already sorted portion of the list will be divided into smaller sorted sublists, and the unsorted portion will be similarly divided and sorted. Then, merging the sorted sublists efficiently combines the elements to produce the final sorted list.

4. Suitability for Large Lists: Merge Sort is efficient for sorting large datasets. Its consistent time complexity and recursive nature allow it to scale well for larger lists, making it an appropriate choice for handling larger datasets with varying levels of sorting.

5. Stability for Partially Sorted Lists: Merge Sort's stability is beneficial when sorting partially sorted lists. Since the algorithm merges smaller sorted sublists, it minimizes unnecessary element swaps and comparisons for already sorted portions, improving overall efficiency.

Considering these factors, Merge Sort is a reliable and efficient choice for sorting a list consisting of a sorted portion followed by random elements. Its stability and consistent time complexity make it well-suited to handle such scenarios with optimal performance.

Question 3

The searching technique that takes O(1) time to find a data is "Direct Access" or "Random Access" search.

Explanation:

In Direct Access or Random Access search, data elements are stored in an array or a similar data structure, and each element has a unique and fixed address or index. When you need to access a specific element, you can directly calculate its address using its index, making the access time constant and independent of the size of the data structure.

Reasons:

1. Constant Time Complexity: Direct Access search has a time complexity of O(1) because the time taken to access an element is constant, irrespective of the number of elements in the data structure. This constant time complexity is achieved by directly calculating the address of the desired element based on its index.

2. No Comparison Required: Unlike other searching techniques such as binary search or linear search, Direct Access search does not involve any comparison between elements. Instead, it directly accesses the element based on its unique index, eliminating the need for iterative or binary comparisons.

3. Deterministic and Predictable: The search time in Direct Access is deterministic and predictable since the address calculation is straightforward and does not depend on any conditional statements or search operations.

4. Suitable for Arrays: Direct Access search is especially suitable for arrays or similar data structures where elements are accessed by their indices. In these data structures, elements are stored contiguously in memory, making it efficient to calculate the address of any element based on its index.

Limitation:

The main limitation of Direct Access search is that it requires a data structure that supports direct access to elements using their indices, like arrays. If the data is not organized in such a structure or if you don't have access to the index directly, then this technique cannot be used. In such cases, other searching techniques like linear search or binary search would be more appropriate, but they may have higher time complexity depending on the size of the data.

Question 4

To sort the given numbers [5, 1, 6, 2, 4] in ascending order using Bubble Sort, we repeatedly swap adjacent elements if they are in the wrong order. The process continues until the list is sorted. Let's illustrate the steps:

Step 1: Original List [5, 1, 6, 2, 4]

- Compare 5 and 1: Swap (1, 5). List becomes [1, 5, 6, 2, 4]. (1 interchange)
- Compare 5 and 6: No swap needed. List remains [1, 5, 6, 2, 4].
- Compare 6 and 2: Swap (2, 6). List becomes [1, 5, 2, 6, 4]. (1 interchange)
- Compare 6 and 4: Swap (4, 6). List becomes [1, 5, 2, 4, 6]. (1 interchange)

Step 2: [1, 5, 2, 4, 6]
- Compare 1 and 5: No swap needed. List remains [1, 5, 2, 4, 6].
- Compare 5 and 2: Swap (2, 5). List becomes [1, 2, 5, 4, 6]. (1 interchange)
- Compare 5 and 4: Swap (4, 5). List becomes [1, 2, 4, 5, 6]. (1 interchange)

Step 3: [1, 2, 4, 5, 6]
- Compare 1 and 2: No swap needed. List remains [1, 2, 4, 5, 6].
- Compare 2 and 4: No swap needed. List remains [1, 2, 4, 5, 6].
- Compare 4 and 5: No swap needed. List remains [1, 2, 4, 5, 6].

The list is now sorted in ascending order: [1, 2, 4, 5, 6].

Number of interchanges required:

In total, 4 interchanges were required to sort the given numbers using Bubble Sort. The algorithm compares adjacent elements in the list and swaps them if they are in the wrong order. As we can see from the steps above, Bubble Sort efficiently sorts the list by repeatedly moving the largest elements to their correct positions.

Question 5

To convert the given expression from infix notation to postfix notation, we can use the Shunting Yard algorithm. The postfix form of the expression (A + B)∗(C∗D − E)∗F / G is as follows:

Postfix Expression: AB+C*CD*E-*F*G/

Question 6

To convert the given prefix expression to postfix notation, we can use the stack-based algorithm. The postfix form of the expression -A/B*C$DE is as follows:

Postfix Expression: AB/-C*DE$-

Question 7

Merging 4 sorted files containing 50, 10, 25, and 15 records will not take O(100) time. Instead, the time complexity for merging the files can be calculated as follows:

Let's assume the number of records in each file is n1, n2, n3, and n4, respectively.

1. Merging two sorted files of sizes n1 and n2 will take O(n1 + n2) time, as we need to compare and merge each element from both files to create a new sorted file.

2. Merging two sorted files of sizes n3 and n4 will take O(n3 + n4) time.

3. Merging the two files created in step 1 and step 2 will take O((n1 + n2) + (n3 + n4)) time.

4. Finally, merging the last two files with sizes (n1 + n2 + n3 + n4) and n5 (if there is another file) will take O((n1 + n2 + n3 + n4) + n5) time.

Therefore, the total time complexity for merging the four sorted files will be:

O((n1 + n2) + (n3 + n4) + ((n1 + n2) + (n3 + n4)) + ((n1 + n2 + n3 + n4) + n5))

In the worst case scenario, the largest file (n1 + n2 + n3 + n4) dominates the time complexity, and the total time complexity for merging the four sorted files will be O(n1 + n2 + n3 + n4).

So, it is not accurate to say that merging 4 sorted files containing 50, 10, 25, and 15 records will take O(100) time. The actual time complexity will depend on the sizes of the individual files being merged.

Question 8

In the worst-case scenario, Quick Sort has a time complexity of O(n^2), not O(n^2/2). Let's explain why.

Quick Sort is a divide-and-conquer sorting algorithm that selects a pivot element from the array and partitions the array into two subarrays, one with elements less than the pivot and the other with elements greater than the pivot. It then recursively applies the same process to the two subarrays until the entire array is sorted.

The time complexity of Quick Sort depends on the choice of the pivot element. In the best and average cases, the pivot is chosen such that it roughly divides the array into two equal-sized partitions. As a result, the algorithm effectively reduces the problem size by half at each recursive call, resulting in a time complexity of O(n log n) for these cases.

However, in the worst-case scenario, the pivot is chosen poorly, causing an imbalanced partitioning of the array. For example, if the pivot is always selected as the smallest or largest element in the array, the partitioning may result in one subarray with n-1 elements and the other subarray with only one element.

In this worst-case scenario, Quick Sort behaves similarly to Selection Sort or Insertion Sort, where it has to sort one subarray at a time with each recursive call. The recurrence relation becomes:

$$T(n) = T(n-1) + T(1) + O(n)$$

The O(n) term represents the time required to partition the array. The recursive calls T(n-1) and T(1) are required to sort the two subarrays.

By solving the recurrence relation, we get a time complexity of O(n^2) for the worst-case scenario of Quick Sort.

So, in summary, Quick Sort has a worst-case time complexity of O(n^2), not O(n^2/2). The worst-case occurs when the pivot selection leads to unbalanced partitioning and resembles the behavior of inefficient quadratic sorting algorithms.

Question 9

Quick Sort is a sorting algorithm that exploits the divide-and-conquer design technique to efficiently sort an array of elements. It divides the input array into smaller subarrays, recursively sorts them, and then combines the sorted subarrays to produce the final sorted output. The algorithm follows these main steps:

1. Divide: Choose a pivot element from the array. The pivot's choice can vary, but common strategies include selecting the first, last, or middle element. The pivot's selection affects the efficiency of Quick Sort in the worst-case scenario.

2. Partition: Rearrange the elements in the array such that all elements smaller than the pivot are on its left, and all elements greater than the pivot are on its right. The pivot is now in its correct sorted position.

3. Conquer: Recursively apply Quick Sort to the subarrays created on the left and right of the pivot. These subarrays are effectively smaller problems of the original array.

4. Combine: As the recursive calls return, the sorted subarrays are combined to produce the final sorted array.

Here is a step-by-step description of how Quick Sort works:

Step 1: Choose a pivot element (P) from the array. Let's assume it is the first element in the array.

Step 2: Partition the array around the pivot:

- Reorder the array so that all elements smaller than P are to its left, and all elements greater than P are to its right.
- Place the pivot P in its correct sorted position.

Step 3: Recursively apply Quick Sort to the subarrays on the left and right of the pivot:

- For the left subarray, repeat steps 1 to 3 with the elements smaller than P.
- For the right subarray, repeat steps 1 to 3 with the elements greater than P.

Step 4: Combine the sorted subarrays:

- The left subarray, the pivot P, and the right subarray are now sorted.
- Concatenate the left subarray, P, and the right subarray to produce the final sorted array.

The recursion in the Quick Sort algorithm continues until the base case is reached, where the subarrays have one or zero elements (which are already sorted by definition). The combination of sorted subarrays during the recursion ultimately leads to a fully sorted array.

Quick Sort's efficiency largely depends on the pivot's selection and the partitioning scheme. In the best and average cases, the pivot selection leads to roughly balanced partitions, resulting in an average time complexity of O(n log n). However, in the worst-case scenario (e.g., always selecting the smallest or largest element as the pivot), the partitions become highly imbalanced, leading to a time complexity of O(n^2). Despite the worst-case scenario, Quick Sort is commonly preferred due to its efficiency in practice and its ability to outperform other sorting algorithms on average.

Question 10

To merge 4 sorted files containing 15, 3, 9, and 8 records into a single sorted file, we can use the "K-way merge" approach, where K is the number of sorted files.

The total number of comparisons required to merge K sorted files can be calculated using the formula: (Total number of records - K)

Let's calculate the total number of comparisons required in this case:

Total number of records = 15 + 3 + 9 + 8 = 35

Number of sorted files (K) = 4

Total number of comparisons = 35 - 4 = 31

So, the total number of comparisons required to merge the 4 sorted files into a single sorted file is 31.

Question 11

The sorting algorithms that do not have a worst-case running time of O(n^2) are:

1. Merge Sort: Merge Sort has a worst-case running time of O(n log n). It is a divide-and-conquer algorithm that consistently divides the input array into halves, resulting in a time complexity that remains efficient even for large datasets.

2. Heap Sort: Heap Sort also has a worst-case running time of O(n log n). It uses a binary heap data structure to efficiently sort elements and has a guaranteed worst-case time complexity.

3. Quick Sort (with a good pivot selection): Quick Sort can achieve an average-case time complexity of O(n log n) with a good pivot selection strategy. Although Quick Sort can have a worst-case time complexity of O(n^2) if the pivot is always chosen poorly, it can be optimized to achieve better performance with various pivot selection methods.

4. Tim Sort: Tim Sort is a hybrid sorting algorithm derived from Merge Sort and Insertion Sort. It has a worst-case time complexity of O(n log n) due to its divide-and-conquer nature combined with insertion-based optimizations.

These sorting algorithms are considered efficient for large datasets because their worst-case time complexity does not degrade to O(n^2) like some other sorting algorithms such as Bubble Sort or Insertion Sort. The worst-case time complexity of O(n log n) allows them to efficiently handle larger datasets, making them commonly used in practice.

Question 12

The complexity of searching an element from a set of n elements using the Binary Search algorithm is O(log n).

Explanation:

Binary Search is a search algorithm that works on sorted arrays or lists. It employs a divide-and-conquer approach to find the target element efficiently. Here's how the Binary Search algorithm works:

1. Start by setting two pointers, "left" and "right," to the first and last elements of the sorted array, respectively.

2. Calculate the middle element index as "mid" using the formula: mid = (left + right) / 2.

3. Compare the target element with the middle element.

- If the target is equal to the middle element, the search is successful, and the index of the target element is returned.

- If the target is less than the middle element, set "right" to mid - 1 to search in the left half of the array.

- If the target is greater than the middle element, set "left" to mid + 1 to search in the right half of the array.

4. Repeat steps 2 and 3 until the target element is found or the search space is empty (left > right).

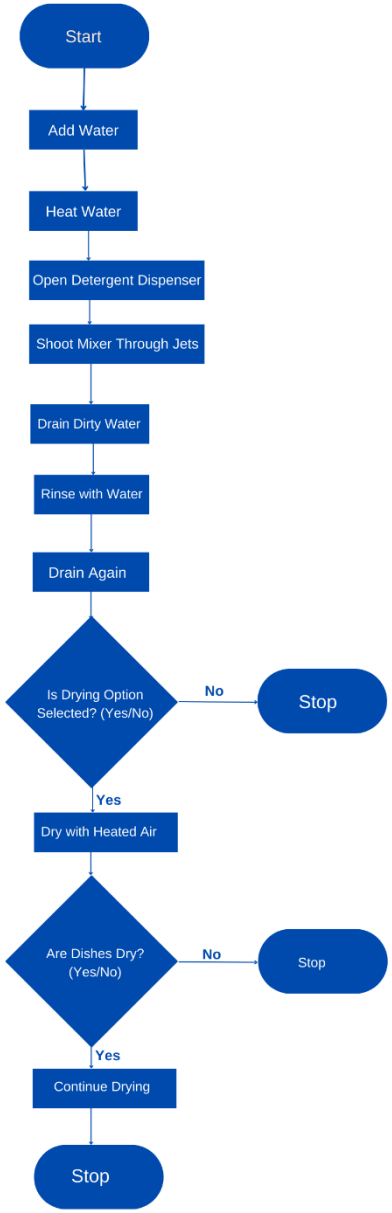
In each iteration of Binary Search, the search space is effectively divided in half. This halving of the search space at each step is what makes the algorithm efficient. As a result, Binary Search eliminates half of the remaining elements in each iteration, leading to a time complexity of O(log n).

In simple terms, with each comparison, Binary Search narrows down the search space to approximately half of the remaining elements. As the number of elements doubles, Binary Search only requires one more comparison to find the target element, making it highly efficient for large

datasets. The logarithmic time complexity (O(log n)) makes Binary Search much faster than linear search (O(n)) for large datasets. However, for Binary Search to work correctly, the input array must be sorted in ascending or descending order.

Question 13

a. Flow Chart for Dishwasher Process:


Start
Add Water
Heat Water
Open Detergent Dispenser
Shoot Mixer Through Jets
Drain Dirty Water
Rinse with Water
Drain Again
Is Drying Option Selected? (Yes/No)
No
Stop
Yes
Dry with Heated Air
Are Dishes Dry? (Yes/No)
No
Stop
Yes
Continue Drying
Stop

b. Algorithm for Dishwasher Process:

Start Dishwasher Process

Step 1: Add Water

- Open the water valve to fill the dishwasher with water to the appropriate level.

Step 2: Heat Water

- Activate the heating element to heat the water to the desired temperature for washing.

Step 3: Open Detergent Dispenser

- At the appropriate time during the cycle, open the detergent dispenser to release the detergent, allowing it to mix with the water.

Step 4: Shoot Mixer Through Jets

- Turn on the jets to spray the detergent-water mixture onto the dishes, cleaning them.

Step 5: Drain Dirty Water

- Open the drain valve to remove the dirty water from the dishwasher.

Step 6: Rinse with Water

- Fill the dishwasher with clean water to rinse off the detergent residue from the dishes.

Step 7: Drain Again

- Drain the rinse water from the dishwasher.

Step 8: Dry with Heated Air (Optional)

- If the user has selected the drying option, activate the heating element to heat the air, which will dry the dishes.

Step 9: Stop

- End the dishwasher process.

End Dishwasher Process

Note: The above algorithm provides a high-level overview of the dishwasher process. In an actual implementation, there may be additional details and conditions to handle various scenarios, such as sensor readings, user settings, safety checks, and error handling. The flow chart and algorithm can be expanded and modified to accommodate these specific requirements.