Design and Analysis of Algorithms

Obtained Marks

BSE 6A

Assignment # 01

Submitted by

Student Name: Muhammad Rehan

Roll # : 22P-9106

Submitted to

Dr. Muhammad Aasim Qureshi

Date assigned: February 14, 2025

Date of submission: February 21, 2025

CS2009 Assignment 01

Muhammad Rehan
22P-9106
BSE-6A

February 18, 2025

Introduction

The assignment provides a concise description of six sorting algorithms. They are Insertion Sort, Merge Sort, Quick Sort, Count Sort, Selection Sort, and Bubble Sort. Each algorithm will be explicated in detail with implementation requirements: input and output specifications, conceptualizations, pseudocodes, and a complete analysis of performance based on characteristics such as time and space complexities, stability, along with other pertinent features. The execution of each of the algorithms on the sample array $[4, 2, 7, 1, 5, 8, 3, 6, 9, 2]$ shall be visualized using flowcharts that specify important steps or iterations. The presence of duplicates (two 2's) guarantees to test for stability, and enough variabilities are introduced to assess their behaviors.

Sample Array

The sample array I will use is:

$$A = [4, 2, 7, 1, 5, 8, 3, 6, 9, 2]$$

1 Insertion Sort

1.1 Input

The Insertion sort expects a one-dimensional array, usually of comparable types such as integers, floats, strings, or anything that has a defined ordering.The input antenna is as one-dimensional array with size n.

1.2 Output

The rearranged output is basically the same array with values organized in ascending order.Thus an input array of $[4, 2, 7]$ would return a sorted array of $[2, 4, 7]$.

1.3 Basic Idea

This allows it to build the sorted portion of the array incrementally, by considering the first element (trivially sorted) and inserting from the elements that were compared with it each into its correct position by bumping those elements larger to the right.

1.4 Algorithm

```
InsertionSort(A, n):
    For i = 1 to n-1:
        key = A[i]
        j = i - 1
        While j >= 0 and A[j] > key:
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key
```

1.5 Complete Analysis

- **Time Complexity:**
  - *Best-case:* The space complexity is linear, that is, $O(n)$, if the array is already sorted. The inner while loop will have one comparison for each element and will not shift elements.
  - *Worst-case:* $O(n^2)$ when the array is in reverse order; each is compared with all the others earlier on which creates an increasing number of shifts to sort the array.
  - *Average-case:* $O(n^2)$ because, on average, it shifts half the number of elements in the sorted portion.
- **Space Complexity:** $O(1)$, this is a in-place algorithm so it only require a constant amount of extra space (e.g., for variables `key` and `j`).
- **Stability:** It is Stable because when inserting an element, it is placed after any equal elements because of the the condition `A[j] > key`, so it make sure their relative order.
- **Other Characteristics:** It is adaptive too (because it is efficient for partially sorted arrays) and online (can sort as elements are received).

1.6 Visualization

The flowchart shows the key insertions for `key=2` (i=1) and `key=1` (i=3):

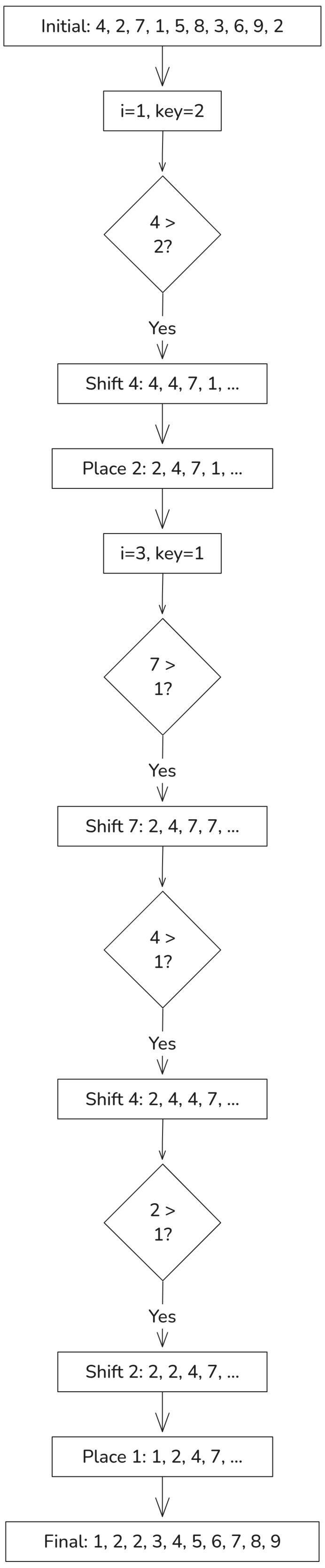

Figure 1: Flowchart showing key insertions

2 Merge Sort

2.1 Input

Merge Sort is an algorithm to sort an array of comparable elements, such as in integers, with no special restrictions on the range or type, other than that they are comparable.

2.2 Output

The output should be an unordered array subjected to some ordering such as ascending.

2.3 Basic Idea

Merge Sort implements the divide-and-conquer methodology: it recursively splits the array into halves, recursively sorts each half, and combines the sorted halves into a single whole in sorted order. The Merge operation combines two sorted subarrays by comparing elements and shuffling them to obey the ordering.

2.4 Algorithm

```
MergeSort(A, left, right):
    If left < right:
        mid = (left + right) / 2
        MergeSort(A, left, mid)
        MergeSort(A, mid + 1, right)
        Merge(A, left, mid, right)

Merge(A, left, mid, right):
    Create temporary arrays L[] and R[] for left and right halves
    Copy A[left..mid] to L[], A[mid+1..right] to R[]
    i = 0, j = 0, k = left
    While i < length(L) and j < length(R):
        If L[i] <= R[j]:
            A[k] = L[i], i++
        Else:
            A[k] = R[j], j++
        k++
    Copy remaining elements of L[] or R[] if any
```

2.5 Complete Analysis

- **Time Complexity:** $O(n \log n)$ for all cases. It do the dividing and takes $O(1)$, and then do the merging of the takes $O(n)$, and there are $\log n$ levels of recursion.
- **Space Complexity:** $O(n)$ because of the auxiliary arrays used in the merge step.

- **Stability:** It could be stable if implemented to prefer the left element when equal (i.e., `L[i] <= R[j]`).
- **Other Characteristics:** It is not in-place, but suitable for large datasets and linked lists, because they could be parallelized.

2.6 Visualization

The flowchart shows the divide and merge process:

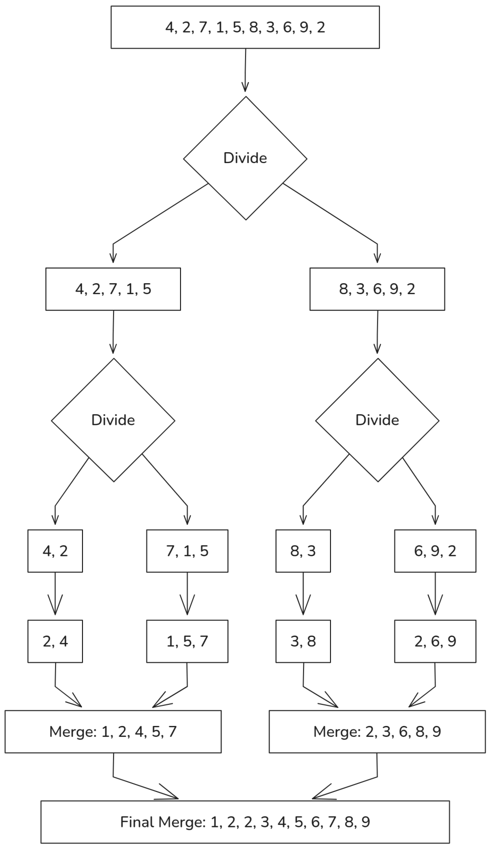

Figure 2: Flowchart showing the divide and merge process

---

3 Quick Sort

3.1 Input

Quick Sort takes an array of comparable elements, normally integers, without a specific constraint.

3.2 Output

The output is the same array reordered in ascending order.

3.3 Basic Idea

Quick Sort uses a divide-and-conquer strategy; that is, a pivot element is chosen, the array is partitioned so that elements less than the pivot go to the left, and those greater go to the right, and the two subarrays are recursively sorted. For simplicity, the Lomuto partition scheme is employed here.

3.4 Algorithm

```
QuickSort(A, low, high):
    If low < high:
        pi = Partition(A, low, high)
        QuickSort(A, low, pi - 1)
        QuickSort(A, pi + 1, high)

Partition(A, low, high): // Lomuto scheme, pivot is A[high]
    pivot = A[high]
    i = low - 1
    For j = low to high - 1:
        If A[j] < pivot:
            i++
            Swap A[i] and A[j]
    Swap A[i + 1] and A[high]
    Return i + 1
```

3.5 Complete Analysis

- **Time Complexity:**
  - *Best-case:* $O(n \log n)$ when the pivot splits the array even two parts.
  - *Worst-case:* $O(n^2)$ when the pivot is the smallest or largest element (e.g., sorted array with last element as pivot).
  - *Average-case:* $O(n \log n)$.
- **Space Complexity:** $O(\log n)$ average for recursion stack, $O(n)$ worst case.
- **Stability:** Not stable, because partitioning can reorder equal elements.
- **Other Characteristics:** In-place (except recursion); its performance kind of depends on pivot choice (randomization mitigates worst-case).

3.6 Visualization

The flowchart illustrates the pivot selection and partitioning process:

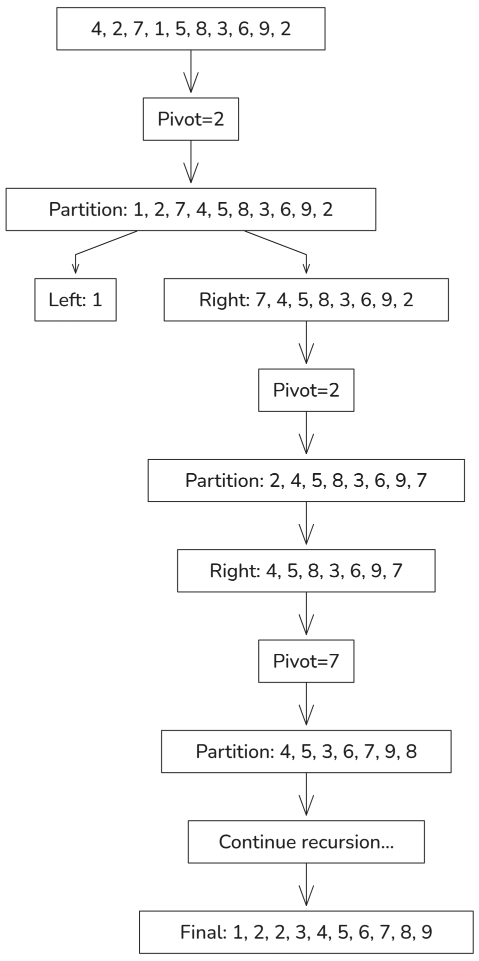
4, 2, 7, 1, 5, 8, 3, 6, 9, 2
Pivot=2
Partition: 1, 2, 7, 4, 5, 8, 3, 6, 9, 2
Left: 1
Right: 7, 4, 5, 8, 3, 6, 9, 2
Pivot=2
Partition: 2, 4, 5, 8, 3, 6, 9, 7
Right: 4, 5, 8, 3, 6, 9, 7
Pivot=7
Partition: 4, 5, 3, 6, 7, 9, 8
Continue recursion...
Final: 1, 2, 2, 3, 4, 5, 6, 7, 8, 9

Figure 3: Flowchart showing the QuickSort pivot selection and partitioning

---

4 Count Sort

4.1 Input

Count Sort has other prerequisites: it only processes non-negative integers for which the maximum known value is k. The smaller the range k spread over the whole array, the more efficient the sorting is.

4.2 Output

The sorted output array is in ascending order.

4.3 Basic Idea

Count Sort counts occurrences of each value and uses these counts to position elements directly into their sorted positions, bypassing the comparison by utilizing integer properties.

4.4 Algorithm

```
CountSort(A, n, k):
    Initialize count[0..k] = 0
    For i = 0 to n-1:
        count[A[i]]++
    For i = 1 to k:
        count[i] += count[i-1]  // Cumulative count
    Initialize output[n]
    For i = n-1 down to 0:
        output[count[A[i]] - 1] = A[i]
        count[A[i]]--
    Copy output to A
```

4.5 Visualization

The flowchart showing the counting and placement steps:

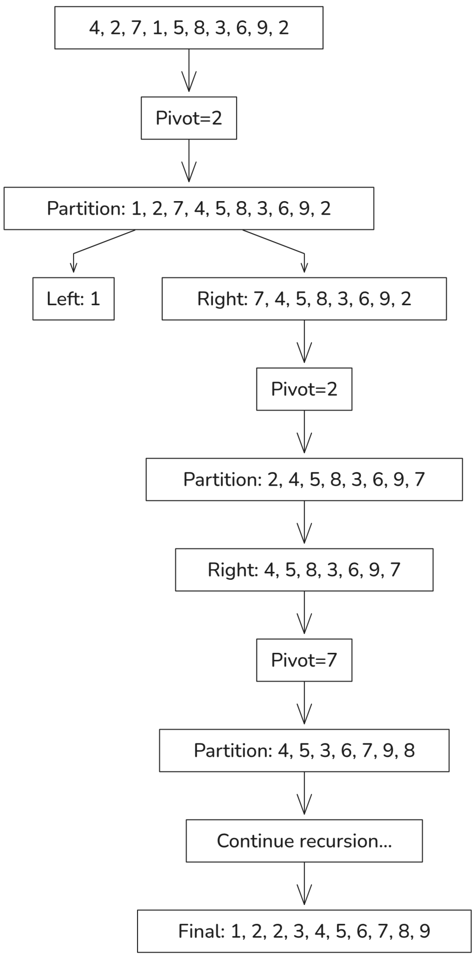
4, 2, 7, 1, 5, 8, 3, 6, 9, 2
Pivot=2
Partition: 1, 2, 7, 4, 5, 8, 3, 6, 9, 2
Left: 1
Right: 7, 4, 5, 8, 3, 6, 9, 2
Pivot=2
Partition: 2, 4, 5, 8, 3, 6, 9, 7
Right: 4, 5, 8, 3, 6, 9, 7
Pivot=7
Partition: 4, 5, 3, 6, 7, 9, 8
Continue recursion...
Final: 1, 2, 2, 3, 4, 5, 6, 7, 8, 9

Figure 4: Flowchart showing counting and placement steps

4.6 Complete Analysis

- **Time Complexity:** $O(n + k)$ for all cases, where k is the range of values.
- **Space Complexity:** $O(n + k)$ for the count and output arrays.
- **Stability:** Stable when placing elements from right to left when we use the cumulative counts.
- **Other Characteristics:** Non-comparison-based, efficient when k is $O(n)$.

---

5 Selection Sort

5.1 Input

Selection Sort is an algorithm that takes an array of comparable elements such as integers.

5.2 Output

The output is the same array arranged in ascending order.

5.3 Basic Idea

Unswapped selection would select the smallest unsorted array element and swap it with the first element of the unsorted portion of the array, such that the selected elements grew the sorted portion of the array one at a time.

5.4 Algorithm

```
SelectionSort(A, n):
    For i = 0 to n-2:
        min_index = i
        For j = i+1 to n-1:
            If A[j] < A[min_index]:
                min_index = j
        Swap A[i] and A[min_index]
```

5.5 Complete Analysis

- **Time Complexity:** $O(n^2)$ for all cases, because it is always scans the entire unsorted portion.
- **Space Complexity:** $O(1)$, in-place.
- **Stability:** Not stable, because swapping can reverse the order of equal elements.
- **Other Characteristics:** Not adaptive; simple but inefficient for large n.

5.6 Visualization

First two passes:

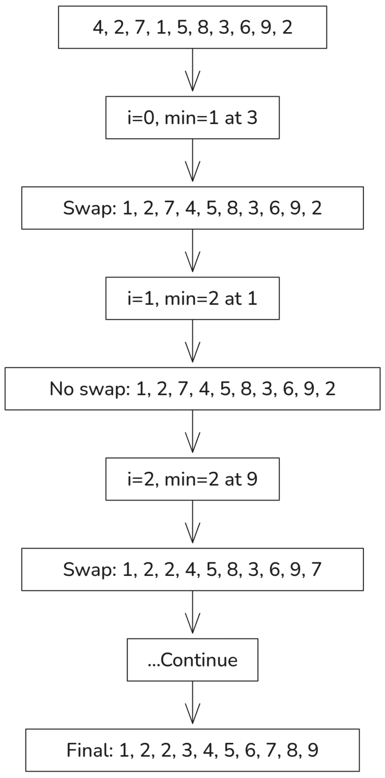
4, 2, 7, 1, 5, 8, 3, 6, 9, 2
i=0, min=1 at 3
Swap: 1, 2, 7, 4, 5, 8, 3, 6, 9, 2
i=1, min=2 at 1
No swap: 1, 2, 7, 4, 5, 8, 3, 6, 9, 2
i=2, min=2 at 9
Swap: 1, 2, 2, 4, 5, 8, 3, 6, 9, 7
...Continue
Final: 1, 2, 2, 3, 4, 5, 6, 7, 8, 9

Figure 5: Flowchart showing the first two passes of Selection Sort

---

6 Bubble Sort

6.1 Input

Bubble Sort accepts an array of comparable elements.

6.2 Output

The output is the same array sorted in ascending order.

6.3 Basic Idea

It checks every pair of adjacent elements and at each iteration it exchanges those of them that are out of order, thus alternating larger objects towards the end bit by bit until the array is ordered correctly.

6.4 Algorithm

```
BubbleSort(A, n):
```

```
For i = 0 to n-2:
    For j = 0 to n-2-i:
        If A[j] > A[j+1]:
            Swap A[j] and A[j+1]
```

6.5 Complete Analysis

- **Time Complexity:**
  - *Best-case:* $O(n)$ with optimization (flag for no swaps); $O(n^2)$ without.
  - *Worst-case:* $O(n^2)$ for reverse order.
  - *Average-case:* $O(n^2)$.
- **Space Complexity:** $O(1)$, in-place.
- **Stability:** Stable, because it only swaps adjacent elements when necessary.
- **Other Characteristics:** Simple but inefficient; adaptive with optimization.

6.6 Visualization

The flowchart showing the first pass of Bubble Sort:

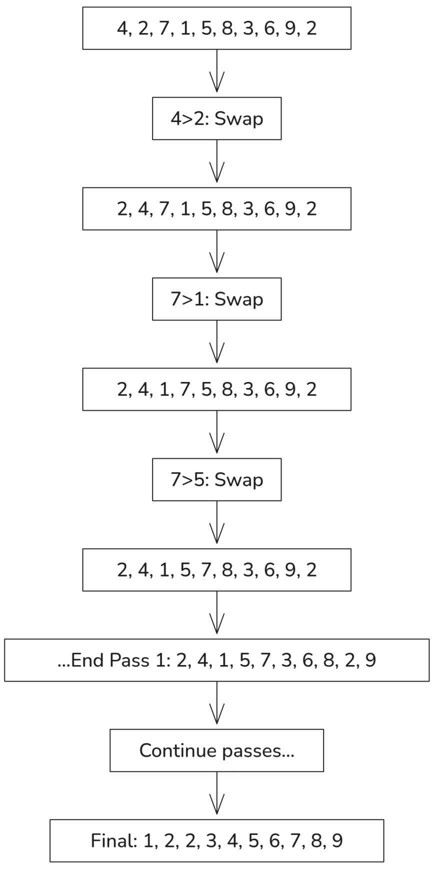

Figure 6: Flowchart showing the first pass of Bubble Sort

7 Question 4

7.1 1.1-1: Real-World Examples

Sorting Example: Consider the library management of returns, books with unique ISBNs returned in random order. The librarian puts the order of the returned books' ISBNs in ascending order to minimize the reshelving time, making the following process efficient-scan the shelves from the lowest to the highest shelf number. Minimal movement times can be achieved, emulating what looks like a sorting algorithm (for example-Merge Sort) that is reliable for large datasets.

Shortest Distance Example: In fact, in a ride-sharing application, the driver needs to pick the nearest passenger. Given the driver's GP coordinates (x_1, y_1) and a set of passenger coordinates (x_i, y_i), the app computes the Euclidean distance for each passenger and identifies the minimum.

$$d = \sqrt{(x_i - x_1)^2 + (y_i - y_1)^2}$$

This is an actual use case to find the shortest distance between two points, usually optimized by making use of spatial data structures like k-d trees.

7.2 1.1-6: Batch vs. Online Input

Think in terms of stock-market trading. In retrospect, when a historical dataset is used for optimizing a portfolio, the trades are simply filled out by time, and the entire data (daily stock prices for another year, concerned only with batch) is available in one place. For instance, using Quick Sort for sorting prices or computing trends. In any active trading scheme, stock prices arrive continuously as a stream. Hence, an online algorithm (modified Insertion Sort, for example) goes through processing every new price as it comes in, dynamically updating the portfolio. The passage emphasizes how batch processing differs from online processing.

7.3 1.2-2: When Does Insertion Sort Beat Merge Sort?

If we suppose that for inputs of size n on a particular computer, insertion sort runs in $8n^2$ steps and merge sort runs in $64n \lg n$ steps. To know when the insertion sort outperforms merge sort, solve the inequality:

$$8n^2 < 64n \lg n.$$

Dividing both sides by $8n$ (with $n > 0$) gives:

$$n < 8 \lg n.$$

Testing sample values:

- For $n = 2$: $2 < 8 \cdot 1 = 8$ (true).
- For $n = 10$: $10 < 8 \cdot 3.32 \approx 26.56$ (true).
- For $n = 40$: $40 < 8 \cdot 5.32 \approx 42.56$ (true).
- For $n = 43$: $43 < 8 \cdot 5.43 \approx 43.44$ (true).
- For $n = 44$: $44 > 8 \cdot 5.46 \approx 43.68$ (false).

Thus, insertion sort beats merge sort for $n \leq 43$.

8 Question 5

8.1 2.1-2: SUM-ARRAY Loop Invariant

Assume the following procedure:

```
SUM-ARRAY(A, n):
    sum = 0
    for i = 1 to n:
        sum = sum + A[i]
    return sum
```

Loop Invariant: At the start of iteration i, the variable `sum` equals $\sum_{k=1}^{i-1} A[k]$.

Step 1: **Initialization:** Before the first iteration ($i = 1$), `sum` is 0, which correctly represents the sum of zero elements.

Step 2: **Maintenance:** Assuming that before iteration i, `sum` equals $\sum_{k=1}^{i-1} A[k]$, after updating `sum = sum + A[i]`, it becomes

$$\sum_{k=1}^{i-1} A[k] + A[i] = \sum_{k=1}^{i} A[k].$$

Step 3: **Termination:** When the loop ends at $i = n + 1$, `sum` equals $\sum_{k=1}^{n} A[k]$, which is returned.

Thus, the procedure correctly computes the sum of $A[1 : n]$.

8.2 2.1-5: ADD-BINARY-INTEGERS Procedure

```
ADD-BINARY-INTEGERS(A, B, n):
    C = new array[0:n]
    carry = 0
    for i = 0 to n-1:
        sum = A[i] + B[i] + carry
        C[i] = sum % 2    // Bit value (0 or 1)
        carry = sum / 2   // Integer division for carry (0 or 1)
    C[n] = carry          // Final carry bit
    return C
```

This procedure is similar to a binary addition: for each bit position i, the sum of $A[i]$, $B[i]$, and some carry is computed, where the least order bit is stored in $C[i]$ and now the carry is updated. In the end, the leftover carry from the last operation is stored in $C[n]$. The time complexity, is $O(n)$.

9 Question 6

9.1 2.2-2: Selection Sort Pseudocode and Analysis

```
SELECTION-SORT(A, n):
    for i = 1 to n-1:
        min_idx = i
        for j = i+1 to n:
            if A[j] < A[min_idx]:
                min_idx = j
        swap A[i] and A[min_idx]
```

Loop Invariant: At the start of iteration i, the subarray $A[1 : i - 1]$ contains the $i - 1$ smallest elements in sorted order.

Why $n - 1$ Iterations? After $n - 1$ iterations, the first $n - 1$ elements are sorted and the final element is naturally the largest.

Running Time:

- *Worst-case:* $\Theta(n^2)$ since the total number of comparisons is $\sum_{i=1}^{n-1}(n - i) = \frac{n(n-1)}{2}$.
- *Best-case:* $\Theta(n^2)$ as the same comparisons are performed regardless of input order.

9.2 2.3-7: Binary Search in Insertion Sort?

Replacing the linear search with binary search in Insertion Sort reduces the search time to $O(\log n)$ per insertion. However, since shifting elements still takes $O(n)$ time per insertion, the overall worst-case running time remains $O(n^2)$.

9.3 2-4: Inversions

a. **Five Inversions in $(2, 3, 8, 6, 1)$:**

$$(2, 1), \quad (3, 1), \quad (8, 1), \quad (8, 6), \quad (6, 1)$$

b. **Maximum Inversions:** The array in descending order, $(n, n-1, \dots, 2, 1)$, has the maximum number of inversions given by

$$\binom{n}{2} = \frac{n(n-1)}{2}.$$

c. **Relationship with Insertion Sort:** Each inversion determines a shift that is needed during insertion; thus, there is a one-to-one correspondence between the number of swaps and the number of inversions, yielding a worst-case time of $O(n^2)$.

d. **Counting Inversions in $\Theta(n \log n)$:** An improved Merge Sort counts inversions during the merge step. If, when merging two subarrays, an element in the left subarray is larger than one in the right subarray, then the number of inversions increases by the count of remaining elements in the left subarray.

10 Question 7

10.1 3-4: Asymptotic Notation Properties

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Evaluate the following statements:

a. **$f(n) = O(g(n))$ implies $g(n) = O(f(n))$:**
False. For example, if $f(n) = n$ and $g(n) = n^2$, then $n = O(n^2)$, but $n^2 \neq O(n)$.

b. **$f(n) + g(n) = \Theta(\min\{f(n), g(n)\})$:**
False. For example, if $f(n) = n^2$ and $g(n) = n$, then $n^2 + n = \Theta(n^2)$, not $\Theta(n)$.

c. **$f(n) = O(g(n))$ implies $\lg f(n) = O(\lg g(n))$:**
True, provided $\lg g(n) \geq 1$ and $f(n) \geq 1$. Since $f(n) \leq c\,g(n)$, it follows that

$$\lg f(n) \leq \lg(c\,g(n)) = \lg c + \lg g(n),$$

which is $O(\lg g(n))$.

d. **$f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$:**
False. For instance, if $f(n) = 2n$ and $g(n) = n$, then $2^{2n} = (2^n)^2$, which is not $O(2^n)$.

e. **$f(n) = O((f(n))^2)$:**
False. For example, if $f(n) = n$, then $n \neq O(n^2)$.

f. **$f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$:**
True. If $f(n) \leq c\,g(n)$, then $g(n) \geq \frac{1}{c} f(n)$.

g. **$f(n) = \Theta(f(n/2))$:**
False. For instance, if $f(n) = n$, then $n \neq \Theta(n/2)$.

h. **$f(n) + o(f(n)) = \Theta(f(n))$:**
True. If $h(n) = o(f(n))$, then $h(n)/f(n) \to 0$, and thus $f(n) + h(n)$ is asymptotically equivalent to $f(n)$.

10.2 3-6: Variations on O and Ω

a. **$f(n) = O(g(n))$ or $f(n) = \widetilde{\Omega}(g(n))$:**
True. If $f(n)$ is not $O(g(n))$, it must eventually exceed $c\,g(n)$ for infinitely many n, satisfying $\widetilde{\Omega}(g(n))$.

b. **Neither O nor Ω holds:**
For example, let $f(n) = n \sin\left(\frac{n\pi}{2}\right)$ and $g(n) = n$. Here, $f(n)$ oscillates and does not have a consistent bound relative to $g(n)$.

c. **Advantages/Disadvantages of $\widetilde{\Omega}$:**
Advantage: It captures functions that exceed $g(n)$ infinitely often.
Disadvantage: It is less precise than Ω when establishing concrete lower bounds.

d. **Impact of O' on Theorem 3.1:**
Defining $f(n) = O'(g(n))$ to allow negative values of $f(n)$ can break the symmetry with Ω (e.g., if $f(n) = -n$ and $g(n) = n$).

11 Question 8

11.1 Two Complex Recursive Functions

1. **Function 1: Tree Path Sum**

```
TREE-PATH-SUM(n):
    if n <= 1:
        return n
    return TREE-PATH-SUM(n-1) + TREE-PATH-SUM(n-2) + n
```

Recurrence:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

with base cases $T(0) = T(1) = O(1)$.
Time Complexity: This recurrence is similar to the Fibonacci recurrence (with an added n term) and leads to exponential time, approximately $O(\phi^n)$ where $\phi = \frac{1+\sqrt{5}}{2}$.

2. **Function 2: Nested Partition**

```
NESTED-PARTITION(n):
    if n <= 1:
        return 1
    sum = 0
    for i = 1 to n/2:
        sum = sum + NESTED-PARTITION(i) * NESTED-PARTITION(n-i)
    return sum
```

Recurrence:

$$T(n) = \sum_{i=1}^{n/2} \Big[T(i) + T(n-i)\Big] + O(n),$$

with base case $T(1) = O(1)$.
Time Complexity: This recurrence is similar in nature to those encountered in Catalan number computations, with an approximate time complexity of

$$T(n) = O\left(\frac{4^n}{n^{3/2}}\right).$$

End of Assignment

Task/Assignment Completion Summary

Task/ Question #	%age Completed	%age Copied	Remarks /Comments
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

I, solemnly declare that the above information is true to the best of my knowledge

[illegible]

Sig.

Name: Muhammad Rehan