1. **Determine whether InsertionSort, BubbleSort, SelectionSort are *stable* sorting algorithms, and in each case, explain your answer.**

A sorting algorithm is considered stable when it maintains the original relative positioning of elements that have identical values. If elements with equal keys appear in a certain sequence in the unsorted data, a stable algorithm will preserve this sequence in the sorted result.

**a) Insertion Sort: Maintains Stability**

**Classification: Stable**

Insertion Sort achieves stability through its incremental approach. As it processes each element, it shifts greater elements rightward while carefully placing the current element in its proper position. This methodology naturally preserves the original order of duplicate values.

**Illustration:**

Initial Array: [(A, 1), (B, 2), (C, 1)]

After Sorting: [(A, 1), (C, 1), (B, 2)]

Notice how A still precedes C in the result, maintaining their original sequence among elements with value 1.

**b) Bubble Sort: Preserves Order**

**Classification: Stable**

Bubble Sort's stability comes from its comparison mechanism. It only exchanges adjacent elements when they're in the wrong order (when the left element is greater than the right). When encountering equal values, no swap occurs, thus preserving their original sequence.

**Illustration:**

Initial Array: [(A, 2), (B, 1), (C, 2)]

After Sorting: [(B, 1), (A, 2), (C, 2)]

A and C (both with value 2) maintain their original relative positions.

**c) Selection Sort: Disrupts Order**

**Classification: Unstable**

Selection Sort compromises stability through its swapping mechanism. It repeatedly identifies the minimum element and exchanges it with the element at the current position, potentially altering the relative positions of equal values.

**Illustration:**

Initial Array: [(A, 2), (B, 1), (C, 2)]

First Pass: Find minimum (B) and swap with first position → [(B, 1), (A, 2), (C, 2)]

Second Pass: Between remaining elements with equal values (2), if C is selected as "minimum" → [(B, 1), (C, 2), (A, 2)]

Final Result: [(B, 1), (C, 2), (A, 2)]

The original sequence of A before C (both with value 2) is reversed in the output.

1. **Perform the MergeSort algorithm by hand on the array [7, 6, 5, 4, 3, 2, 1]. Show all steps, in the way that was done in the lecture.**

**Steps:**

1. Initial Array: [7, 6, 5, 4, 3, 2, 1]
2. Split into two halves: [7, 6, 5] and [4, 3, 2, 1].

For [7, 6, 5]:

* + Split into [7] and [6, 5].
  + [6, 5] → [6] and [5].
  + Merge [6] and [5] → [5, 6].
  + Merge [7] and [5, 6] → [5, 6, 7].

For [4, 3, 2, 1]:

* + Split into [4, 3] and [2, 1].
  + [4, 3] → [4] and [3], merge → [3, 4].
  + [2, 1] → [2] and [1], merge → [1, 2].
  + Merge [3, 4] and [1, 2] → [1, 2, 3, 4].

1. Merge [5, 6, 7] and [1, 2, 3, 4]:
   * Final result: [1, 2, 3, 4, 5, 6, 7].
2. **Sometimes MergeSort is supplemented with a secondary sorting routine (typically, InsertionSort is used) in the following way: During the recursion in MergeSort, the size of the array being sorted becomes smaller and smaller. To create a hybrid sorting routine, then a recursive call requires the algorithm to process an array with 20 or fewer elements, give this array to InsertionSort and patch in the result after it has finished. Call this hybrid algorithm MergeSortPlus.** 
   * 1. **Express the steps of MergeSortPlus in the pseudo-code language we are using in class.**

MergeSortPlus(arr, left, right):

if (right - left + 1) ≤ 20:

InsertionSort(arr, left, right)

else:

mid = (left + right) / 2

MergeSortPlus(arr, left, mid)

MergeSortPlus(arr, mid + 1, right)

Merge(arr, left, mid, right)

* + 1. **Write the Java code for MergeSortPlus (use the implementation of MergeSort provided in the lab folder)**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer

AI-generated content may be incorrect.**

**C. Run tests to compare running times of MergeSort and MergeSortPlus. Which one runs faster? Explain how you tested and whether you feel your results are conclusive.**

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

Based on performance testing, MergeSortPlus demonstrates superior speed compared to standard MergeSort. To evaluate these algorithms objectively, I implemented both in Java and conducted comparative analysis using three different input sizes: small (10 elements), medium (100 elements), and large (1000 elements). The methodology was carefully controlled - identical random integer arrays were created for each test case, with copies used to ensure both algorithms processed exactly the same data. Performance measurement utilized System.nanoTime() to capture precise execution times before and after each sorting operation. Multiple test runs were conducted across all three array sizes to establish consistent performance patterns. The results consistently showed that MergeSortPlus outperformed traditional MergeSort, validating the effectiveness of its hybrid approach. This empirical evidence supports the theoretical benefits of the optimization strategies implemented in MergeSortPlus, demonstrating measurable efficiency gains particularly evident in larger datasets.

1. ***Binary Trees.* A *binary tree* is a tree in which every node has at most two children.**

**a. Write out 4 different binary trees, each having height = 3 – make sure that no**

**two of your trees have the same number of nodes. (There is no need to give**

**labels to the nodes.)**

**Tree one:**

A

/ \

B C

/ \ / \

D E F G

/ \ / \ / \ / \

H I J K L M N O

**Tree Two:**

1

/ \

2 3

/ \ /

4 5 6

\ \

7 8

**Tree Three:**

1

/

2

/

3

/

4

**Tree Four:**

**A**

**/ \**

**B C**

**/ \ \**

**D E F**

**/**

**G**

**b. Examine the trees you have drawn and decide whether the following**

**statement is true or false:**

***Every binary tree of height 3 has at most 23=8 leaves.***

True.

A leaf node is defined as a node that has no children.

For a binary tree with height h=3, the maximum number of leaves is achieved in a full binary tree where every level is completely filled. In such a tree, the number of leaves is 2^3 = 8.

Any binary tree with an imbalanced structure, like trees two, three, and four, will always contain fewer than 8 leaf nodes. These imbalanced trees have some internal nodes with only one child or missing children, which reduces the total number of possible leaf nodes compared to a perfectly balanced full binary tree of the same height.

**c. Based on your answer to b, what do you think is true in general about the number of leaves of a binary tree of height *n*?**

The maximum number of leaves in a binary tree of height n is 2^n.

This maximum leaf count occurs only in a perfect binary tree where every internal node has exactly two children and all leaves are at the same depth. Such a tree has a complete structure where every level contains the maximum possible number of nodes.

In binary trees that aren't perfectly filled (those with missing nodes or incomplete branches), the leaf count will always be fewer than 2^n. This reduction happens because some potential leaf positions remain unfilled when branches terminate earlier than the maximum height.