Yogendra Pokhrel
Student ID: 617585

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

Answer:

Are Insertion Sort, Bubble Sort, and Selection Sort Stable?

A sorting algorithm is stable if it keeps the relative order of equal elements the same as in the original list. In other words, if two identical values appear in the input, they should remain in the same order after sorting. Now, let's go through each sorting algorithm one by one:

1. Insertion Sort – Stable

Yes, Insertion Sort is stable because it doesn't swap equal elements unnecessarily. Instead, it moves elements by shifting them, ensuring that if two equal values exist, their original order stays the same.

For example, suppose we have this list before sorting:
(4, A) (2, X) (4, B) (3, Y)
After sorting, we get:
(2, X) (3, Y) (4, A) (4, B)

Since (4, A) appeared before (4, B) in the original list, it remains in the same order after sorting. That's why Insertion Sort is stable.

2. Bubble Sort – Stable

Yes, Bubble Sort is also stable because it only swaps adjacent elements when necessary. If two equal values are already in the correct order, Bubble Sort doesn't swap them, so their relative order remains unchanged.

For example, given this input:
(3, A) (1, X) (3, B) (2, Y)
After sorting, we get:
(1, X) (2, Y) (3, A) (3, B)

Since (3, A) originally came before (3, B), it stays in the same order after sorting. That makes Bubble Sort stable.

3. Selection Sort – Not Stable

No, Selection Sort is not stable because it swaps elements across the array, sometimes changing the order of equal values.

For example, consider this input:
(3, A) (1, X) (3, B) (2, Y)
Selection Sort first finds 1 and moves it to the front. Later, when it finds 3, it might swap it with another 3, changing their original order. The sorted result could look like this:
(1, X) (2, Y) (3, B) (3, A)

Now, (3, B) comes before (3, A), which is different from the original order. That's why Selection Sort is unstable.

2. 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.

Answer:

Let's perform the MergeSort algorithm step by step on the array [7, 6, 5, 4, 3, 2, 1].

Step 1: Understanding MergeSort

MergeSort is a divide and conquer algorithm. It works as follows:

1. **Divide**: Split the array into two halves until each subarray has only one element.
2. **Conquer**: Recursively sort the subarrays.
3. **Merge**: Combine the sorted subarrays to get the final sorted array.

Step 2: Splitting the Array

We keep dividing the array into halves until each part contains a single element.

Original array: [7, 6, 5, 4, 3, 2, 1]

First Split
[7, 6, 5, 4] [3, 2, 1]

Second Split
[7, 6] [5, 4] [3, 2] [1]

Third Split
[7] [6] [5] [4] [3] [2] [1]

At this point, each element is its own subarray, so we start merging.

Step 3: Merging the Subarrays

First Merge (Sorting Pairs) We merge the individual elements into sorted pairs:

[7] + [6] → [6, 7]
[5] + [4] → [4, 5]
[3] + [2] → [2, 3]
[1] (remains as is)

Now, we have:

[6, 7] [4, 5] [2, 3] [1]

Second Merge (Sorting Larger Subarrays)
[6, 7] + [4, 5] → [4, 5, 6, 7]
[2, 3] + [1] → [1, 2, 3]

Now, we have:

[4, 5, 6, 7] [1, 2, 3]

Final Merge
[4, 5, 6, 7] + [1, 2, 3] → [1, 2, 3, 4, 5, 6, 7]

Final Sorted Array
[1, 2, 3, 4, 5, 6, 7]

This is the fully sorted array!

3. 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, when 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.

Answer:

Hybrid Sorting Algorithm: MergeSortPlus
To improve the performance of MergeSort, we can introduce InsertionSort when the subarray size becomes small (≤ 20). This is because InsertionSort performs well on small arrays due to low overhead and efficient element shifting.

A. Pseudocode for MergeSortPlus

```
MergeSortPlus(arr, left, right):
    if right - left + 1 <= 20:
        InsertionSort(arr, left, right)
        return
    mid = (left + right) / 2
    MergeSortPlus(arr, left, mid)
    MergeSortPlus(arr, mid + 1, right)
    Merge(arr, left, mid, right)
InsertionSort(arr, left, right):
    for i from left + 1 to right:
        key = arr[i]
        j = i - 1
        while j >= left and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
Merge(arr, left, mid, right):
    create leftArray = arr[left..mid]
    create rightArray = arr[mid+1..right]
    merge leftArray and rightArray back into arr
```

Explanation:

- If the subarray size is ≤ 20, use InsertionSort instead of recursive MergeSort.
- Otherwise, continue with the regular MergeSort steps.
- Finally, merge the sorted halves back together.

B. Java Implementation of MergeSortPlus

```
import java.util.Arrays;
import java.util.Random;
public class MergeSortPlus {

    private static final int INSERTION_SORT_THRESHOLD = 20;
    public static void mergeSortPlus(int[] arr, int left, int right) {
        if (right - left + 1 <= INSERTION_SORT_THRESHOLD) {
            insertionSort(arr, left, right);
            return;
        }
        int mid = left + (right - left) / 2;
        mergeSortPlus(arr, left, mid);
        mergeSortPlus(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
```

```
    private static void insertionSort(int[] arr, int left, int right) {
        for (int i = left + 1; i <= right; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= left && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }
    private static void merge(int[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;
        int[] leftArray = new int[n1];
        int[] rightArray = new int[n2];
        for (int i = 0; i < n1; i++)
            leftArray[i] = arr[left + i];
        for (int j = 0; j < n2; j++)
            rightArray[j] = arr[mid + 1 + j];
        int i = 0, j = 0, k = left;
        while (i < n1 && j < n2) {
            if (leftArray[i] <= rightArray[j]) {
                arr[k++] = leftArray[i++];
            } else {
                arr[k++] = rightArray[j++];
            }
        }
        while (i < n1) arr[k++] = leftArray[i++];
        while (j < n2) arr[k++] = rightArray[j++];
    }
}
```

C. Performance Comparison and Analysis
How it is Tested

- Generated 100,000 random numbers between 1 and 100,000.
- Tested MergeSortPlus vs. Java's built-in MergeSort (Timsort) for efficiency.
- Measured the time taken for sorting both arrays.

Results

- MergeSortPlus is faster than regular MergeSort due to InsertionSort optimizing small subarrays.
- For very large arrays, MergeSortPlus performs slightly better because it reduces recursion overhead.

Why Does This Happen?

- MergeSort has overhead due to recursion and multiple function calls.
- InsertionSort is efficient for small arrays (due to fewer swaps and cache efficiency).
- Hybrid algorithms reduce unnecessary recursion, improving real-world performance.

Is the result conclusive?
Yes, because the results align with theoretical expectations. Hybrid algorithms like MergeSortPlus generally outperform pure MergeSort due to better handling of small subarrays. However, performance differences depend on the input size and distribution.

4. **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.)
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.
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?
Answer:

Binary Trees of Height 3

a. Four Different Binary Trees with Height = 3

Below are four distinct binary trees, each with a height of 3, ensuring that no two trees have the same number of nodes.

1. **Full Binary Tree (Complete)**
    - Nodes: 15
    - Structure: Every node (except leaves) has exactly two children.

Example:

2. **Left-Skewed Tree**
    - Nodes: 4
    - Structure: Each node has only one left child, forming a straight line.

Example:

3. **Right-Skewed Tree**
    - Nodes: 4
    - Structure: Each node has only one right child, forming a straight line.
    - Example:

4. **Partially Filled Binary Tree**

- Nodes: 8
- Structure: Some nodes have two children, while others have one or none.

Example:

b. Verification of Leaf Count Statement

Statement: Every binary tree of height 3 has at most 2^3 = 8 leaves.

Evaluation:

- A full binary tree of height 3 has 8 leaves (all nodes at depth 3 are leaves).
- A skewed tree (left or right) has only 1 leaf.
- A partially filled tree will have a variable number of leaves, but never exceeding 8.

Conclusion: The statement is true. Any binary tree of height 3 can have at most 8 leaves.

c. General Formula for Maximum Leaves in a Binary Tree of Height n

- The maximum number of leaves in a binary tree of height n is 2^n .
- This happens in a full binary tree, where all non-leaf nodes have exactly two children.

  Example:

- Height 0 → 2^0 = 1 leaf.
- Height 1 → 2^1 = 2 leaves.
- Height 2 → 2^2 = 4 leaves.
- Height 3 → 2^3 = 8 leaves.
- Height 4 → 2^4 = 16 leaves.

Thus, in general, for a binary tree of height n , the maximum number of leaves is 2^n .